Edward Yourdon

TECHNIQUES OF PROGRAM STRUCTURE AND DESIGN

Techniques of Program Structure and Design

EDWARD YOURDON

President
YOURDON inc.

PRENTICE-HALL, INC., *Englewood Cliffs, New Jersey*

Library of Congress Cataloging in Publication Data

YOURDON, EDWARD.
Techniques of program structure and design.

Includes bibliographies.
1. Electronic digital computers—Programming.
I. Title.
QA76.6.Y68 001.6'42 75-9728
ISBN 0-13-901702-X

10 9 8 7 6 5 4

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA (PTE.) LTD., *Singapore*

PREFACE

"We build systems like the Wright brothers built airplanes—build the whole thing, push it off a cliff, let it crash, and start over again."

Professor R. M. Graham
Software Engineering, page 17

"Of course 99 percent of computers work tolerably well, that is obvious. There are thousands of respectable Fortran-oriented installations using many different machines and lots of good data processing applications running quite steadily; we all know that! The matter that concerns us is the sensitive edge, which is socially desperately significant."

Professor J. N. Buxton
Software Engineering, page 119

"I think it is inevitable that people program, and will continue to program, poorly. Training will not substantially improve matters. Using subsets of languages doesn't help because people always step outside the subset. We have to learn to live with it."

Professor A. J. Perlis
Software Engineering Techniques, page 33

"There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance by software, between what seems to be ultimately

Preface opening quotes from *Software Engineering*, P. Naur and B. Randell, (eds.), NATO Scientific Affairs Division, Brussels 39, Belgium, January 1969 and *Software Engineering Techniques*, J. N. Buxton and B. Randell, (eds.), NATO Scientific Affairs Division, Brussels 39, Belgium, April 1970.

> possible and what is achievable now and between estimates of software costs and expenditures. This gap is arising at a time when the consequences of software failures in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people, and ultimately for nations as well."
>
> Dr. E. E. David and Mr. A. G. Fraser
> *Software Engineering*, page 120

In 1970 I made the terrible mistake of writing a set of notes for a seminar entitled ADVANCED PROGRAMMING TECHNIQUES. I say "mistake" because I quickly discovered that I knew almost nothing about that advanced state of witchcraft that we call programming, despite the fact that I had been gainfully employed in the field for several years. Nevertheless, I persisted: from 80 pages of almost incoherent scribblings in 1970, the notes went through ten major revisions and slowly grew to 900 typewritten pages. At several points in this process, common sense dictated that I throw all ten versions of the manuscript away and cease inflicting my ideas upon my students; unfortunately, pride and sheer orneriness have prevailed. Pity the poor students who suffered through this period: Nearly 3000 programmers in 12 countries have forgiven my bugs and overlooked some of my more absurd ideas; more important, they have all patiently communicated to me what *they* know about programming—a privilege that must be experienced to be appreciated.

It has been especially interesting to me during this period to watch the transformation through which the computer field has been struggling. The emergence of such prophets as Edsger Dijkstra, Niklaus Wirth, and Gerald Weinberg encourages one to think that perhaps programmers will eventually be taught to write good programs from the very beginning.

If this turns out to be the case, some of the suggestions and warnings in this book may turn out to be unnecessary. Both in my seminars and in this book, I have made the basic assumption that the student has been exposed to some of the rudiments of programming, but that he has not been exposed to any significant ideas on "good" programming. I think this is a very reasonable assumption when one is dealing with the vast majority of programmers working in industry. As long as basic training courses continue to stress the mechanics of a particular programming language (as is usually the case with most FORTRAN and COBOL courses) this will continue to be true. It is particularly encouraging to see the recent trend in university courses toward teaching students the elements of good programming.

In any case, that is what this book attempts to discuss: "good" programming. However, my experience has been that it is very difficult to tell someone how to design a good program if he (or she) violently disagrees with me as to what a good program is. Hence, the first chapter in the book is a discussion of the characteristics of a good program. I must admit that

this chapter is aimed at the experienced programmer; a college freshman would probably not have any preconceived ideas about the relative merits of maintainability, flexibility, and efficiency in a program.

The next logical question is: How do we go about designing such a good program? Chapter 2 provides answers in the form of *top-down design*. There seems to be some controversy concerning the order in which some of these ideas are presented. Many people argue that the concept of structured programming should be presented first, after which the programmer is more likely to comprehend and accept the principle of top-down design. Perhaps this is so; indeed, when I am confronted with a group of particularly impatient programmers in a seminar, I find it helpful to present the more tangible concepts of structured programming first, before moving on to the more abstract concept of top-down design. Nevertheless, from a logical point of view, it makes much more sense to talk about the *design* of a program before one talks about a coding discipline.

Chapter 3 contains a discussion of modular programming. It represents a nice transition from the abstract discussion of top-down design to the more detailed discussion of structured programming. Nearly everyone seems to consider modular programming the precursor to the currently fashionable structured programming. I find it ironic that when I began these notes in 1970, modular programming was considered somewhat radical by many programmers; in today's world of structured programming, modular programming is considered passé. Oddly enough, it is just this area of modularity that will probably see the greatest advances in the next few years. Larry Constantine's paper on "Structured Design" in the May 1974 *IBM Systems Journal* will probably be as responsible for generating new interest in modular design as Terry Baker's article in the January 1972 *IBM Systems Journal* was in the field of structured programming.

In any case, structured programming certainly deserves ample discussion in any modern textbook on programming; Chapter 4 discusses this topic at length. One section of the chapter deserves special mention: Section 4.3.3 discusses the conversion of unstructured programs into structured programs. A number of people have objected to the emphasis I have placed on this area, but I submit that the usefulness of this material is a direct function of the students' programming experience. A new programmer, with no previous exposure to GO-TO-ridden "rat's nest" programs, probably requires none of the material in Section 4.3.3—indeed, it may actually be harmful! An experienced programmer, on the other hand, is in desperate need of these conversion techniques, because they help him convert his unstructured *thinking* into structured *thinking*. It is somewhat naive to argue that the programmers currently being trained in universities will learn structured programming from the beginning; that is roughly comparable to the argument that, since FORTRAN lacks the control structures required to conveniently implement structured programs, people should instantly cease

programming in FORTRAN. Things don't work that way in the real world. For better or for worse, people are going to continue programming in FORTRAN for the next several years until we can begin to shift them towards more civilized languages. In the meantime, it is eminently practical to teach people how to accomplish some reasonable approximation of structured programming in FORTRAN. By the same token, we currently have several *hundred thousand* experienced programmers in the world, each of whom probably has another twenty years of "rat's nest" programming before he drops from exhaustion. If we wait for the newly trained university students to rectify the situation with the magic of structured programming, most of our current computer systems will collapse.

After all this preaching about structured programming, I must admit that I temporarily ran out of energy. Chapters 5 and 6—a discussion of programming style and defensive programming—probably do not get the attention they deserve. I offer the following excuse: If the philosophies in Chapters 1–4 are followed faithfully, there is a reasonable chance that the programmer will already have accomplished what is suggested in Chapters 5 and 6. Perhaps a more substantial excuse is that a thorough treatment of programming style would require a book in itself. I highly recommend both *The Elements of FORTRAN Style* (Schneiderman and Kreitzberg; Harcourt, Brace, Jovanovich, 1971) and *The Elements of Programming Style* (Kernighan and Plauger, McGraw-Hill, 1974) in this area.

For various reasons, I felt that a discussion of testing and debugging should accompany a discussion of top-down design, modular programming, and structured programming; hence Chapters 7 and 8. I still insist on making a distinction between testing and debugging, despite the feeling of many of my students that I am belaboring the issue. I must admit to a sense of frustration at the end of the chapter on testing: I have the feeling that the majority of programmers don't really know how to test a program properly. Most computer scientists probably agree that a great deal more work needs to be done before we have achieved the same level of discipline in testing that we have in structured programming. Maybe we need "structured testing?"

If structured testing, why not structured debugging? The major objective of Chapter 8 has been to present some debugging strategies that are quite distinct from the usual approach of dumps and traces. I have always felt that these strategies represented the *art* of debugging, and that if some students found it difficult to apply the strategies, it was because they lacked the inborn artistic talents required of a good debugger. Perhaps this is an oversimplified view; some friends of mine at Shell Oil in Melbourne tell me that their programmers are being given courses in general problem-solving in an attempt to improve their debugging skills. Perhaps this is the beginning of structured debugging.

At the end of the book I have included four major programming exercises that may be used to illustrate many of the principles of program design. The problems in Appendix A and Appendix B were intended to be designed by a *team* of programmers; a group of three or four seems to be the optimal size. Appendix C and Appendix D are small enough to be attacked by an individual programmer.

Acknowledgments

In addition to the thousands of students who read through the manuscript of this book and suggested improvements and corrections, I would like to give special thanks to Brian Kernighan of Bell Labs and Peter Neely of the University of Kansas for their thorough review. My colleagues Bill Plauger, Trish Sarson, and Bob Abbott also deserve thanks for using the material in several training courses and giving me recommendations for improvements in the manuscript. During the production of the book, Lynne Sadkowski retained her composure while continually trying to track me down in some far-distant part of the world to review critical galleys, while Wendy Eakin provided invaluable assistance by copyediting and proofreading my manuscript to the point where it resembled proper English.

In the final analysis, though, I owe this book to Sam, who, more than all the others, believed in me.

CONTENTS

PREFACE xi

1 THE CHARACTERISTICS OF A "GOOD" COMPUTER PROGRAM 1

1.0 Introduction, *2*
1.1 What Are the Qualities of a Good Programmer?, *3*
1.2 What Are the Qualities of a Good Program?, *6*
1.3 Some Concluding Remarks About the "Goodness" of Computer Programs, *28*
Problems, *29*

2 TOP-DOWN PROGRAM DESIGN 36

2.0 Introduction, *36*
2.1 Top-Down Design, *38*
2.2 Top-Down Coding, *54*
2.3 Top-Down Testing, *59*
2.4 Alternatives, Variations, and Problems with Top-Down Design, *77*

2.5 Case Studies and Examples: The IBM "Chief Programmer Team" Project, *83*
References, *88*
Problems, *88*

3 MODULAR PROGRAMMING **93**

3.0 Introduction, *93*
3.1 Definitions of Modularity, *94*
3.2 The Advantages and Disadvantages of Modularity, *97*
3.3 Techniques for Achieving Modular Programs, *99*
3.4 General-Purpose Subroutines, *125*
References, *131*
Problems, *132*

4 STRUCTURED PROGRAMMING **136**

4.0 Introduction, *136*
4.1 History and Background of Structured Programming, *137*
4.2 The Objectives and Motivation of Structured Programming, *140*
4.3 Theory and Techniques of Structured Programming, *144*
4.4 Other Aspects of Structured Programming, *174*
4.5 Considerations of Practicality in Structured Programming, *175*
References, *180*
Problems, *183*

5 PROGRAMMING STYLE: SIMPLICITY AND CLARITY **195**

5.0 Introduction, *196*
5.1 Review of Suggestions for Developing Simple Programs, *197*
5.2 Additional Programming Techniques for Readable Programs Problems, *207*
References, *216*
Problems, *216*

6 ANTIBUGGING 220

6.0 Introduction, *220*
6.1 The Arguments Against Antibugging, *221*
6.2 Aspects of a Computer Program that Require Checking, *224*
6.3 Antibugging Programming Techniques, *228*
Problems, *239*

7 PROGRAM TESTING CONCEPTS 243

7.0 Introduction, *244*
7.1 Definitions and Concepts, *244*
7.2 The Scope of the Testing Problem, *247*
7.3 Levels of Testing Complexity, *249*
7.4 Types of Errors to be Exposed by Testing, *254*
7.5 Stages of Testing, *256*
7.6 Designing Programs for Easier Testing, *263*
7.7 Automated Testing Techniques, *266*
7.8 Other Testing Techniques, *272*
References, *274*
Problems, *275*

8 DEBUGGING CONCEPTS AND TECHNIQUES 279

8.0 Introduction, *279*
8.1 Debugging Philosophies and Techniques, *280*
8.2 Common Programming Errors and Bugs, *292*
8.3 Classical Debugging Techniques, *294*
8.4 The DDT Debugging Packages, *298*
8.5 Implementation of a Simple Version of DDT, *308*
References, *321*
Problems, *322*

APPENDICES: CLASS PROBLEMS AND EXERCISES 326

Introduction, *326*
A The Money Problem, *329*
B The Mugwump Fertilizer Problem, *335*
C The Master-File Update Problem, *346*
D The Tic-Tac-Toe Problem, *350*

INDEX 351

1

THE CHARACTERISTICS OF A "GOOD" COMPUTER PROGRAM

In fact, the central concept in all software is that of a program, and a generally satisfactory definition of a program is still needed. The most frequently used definition—that a program is a sequence of instructions—forces one to ignore the role of data in the program. A better definition is that a program is a set of transformations and other relationships over sets of data and container structures. At least this definition guides the designer to break up a program design problem into the problems of establishing the various data and container structures required, and defining the operators over them. The definition requires that attention be given to the properties of the data regardless of the containers (records, words, sectors, etc.).

KENNETH K. KOLENCE
Software Engineering, page 50

There is no theory which enables us to calculate limits on the size, performance or complexity of software. There is, in many instances, no way even to specify in a logically tight way what the software product is supposed to do, or how it is supposed to do it.

EDWARD E. DAVID
Software Engineering, page 69

One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one

Chapter opening quotes from *Software Engineering*, P. Naur and B. Randell (eds.), NATO Scientific Affairs Division, Brussels 39, Belgium, January 1969 and *Software Engineering Techniques*, J. N. Buxton and B. Randell (eds.), NATO Scientific Affairs Division, Brussels 39, Belgium, April 1970.

thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as a sum of many sub-assemblies.

A. G. FRASER
Software Engineering, page 86

Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

KENNETH K. KOLENCE
Software Engineering, page 123

Any large program will exist during its lifetime in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program, but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program therefore should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstrated of the shared components but also of the shared substructure.

E. W. DIJKSTRA
Software Engineering Techniques, page 31

1.0 Introduction

Throughout this book, we assume that you are already familiar with the basic elements of computer hardware, operating systems, and programming languages; you are now ready to direct your attention to the finer points of computer programming. Which area of programming would you like to explore first? Shall it be list structures? Dynamic memory allocation? Decision tables? Or perhaps searching and sorting algorithms?

If you chose *any* of these areas, you may be suffering from a weakness common to almost all programmers: a fatal fascination with *techniques* of programming. You may be assured that we will eventually discuss a number of important programming techniques, but only after we agree on some *philosophies* of programming.

Philosophical discussions of this type are generally unpopular with programming students; they seem too vague and general, and the programmers would prefer to spend their time discussing more "practical," "useful" subjects. On the other hand, it should be remembered that programmers are, in many cases, a somewhat stubborn, unrealistic, and uncompromising breed: They often seem to think that their primary function is to invent clever new algorithms, rather than to perform useful work.

Without meaning to be unnecessarily cruel, I feel that I must remind you of an important fact: *As a programmer, you will always be working for*

an employer. Unless you are very rich and very eccentric, you will not enjoy the luxury of having a computer in your own home; unless you plan to remain a perpetual student, you cannot expect to spend your life dazzling your awe-stricken professors with your programming virtuosity. In short, you cannot be an artist, separate and aloof; it is highly unlikely that the programming community will ever generate a Michelangelo or Rembrandt. As a programmer, you will be expected to do whatever is necessary to make the computer provide some useful service. This is true whether you are a business programmer, a scientific programmer, or involved in research activities for a computer manufacturer.

I thus feel that it is eminently practical to discuss some philosophies of programming. What things should you do to make yourself useful as a programmer? What aspects of your programs will be of most concern to your employer? These topics may seem rather trifling to you, but they can ensure your success as a programmer. To ignore the philosophies presented in this chapter will certainly cost you dearly in terms of promotions and salary increases, and it may even cost you your job. What could be more practical?

The basic purpose of this chapter is to present a checklist of good program design. We will begin by making some general comments on the qualities of a good *programmer*, as well as the qualities of a good *program*; this will be followed by a more specific list of seven characteristics of a good program. After writing a computer program, you should ask yourself if your program satisfies these "rules" of good programming; *before* writing a program, you should ask yourself how you can best satisfy these rules.

1.1 What Are the Qualities of a Good Programmer?

During the past several years, I have had the opportunity to teach advanced programming courses to thousands of students in several different countries around the world. The students, in general, have been experienced programmers in banks, insurance companies, government agencies, manufacturing organizations, scientific installations, universities, and every other conceivable background. More for my own amusement than anything else, I have often begun each course with the question, "What are the qualities of a good programmer?" The answers have been as varied as the students' backgrounds, and some of them are worth repeating:

1. A good programmer writes good programs (or efficient programs, or well-documented programs, etc.).
2. A good programmer works well with other people.
3. A good programmer communicates well with the users of his program.
4. A good programmer takes a bath at least once a week.
5. A good programmer shows up for work on time.
6. A good programmer *never* shows up for work on time.

7. A good programmer doesn't cause trouble.
8. A good programmer works well under pressure.
9. A good programmer likes classical music.

An argument often breaks out about the first answer to the question, i.e., the statement that a good programmer is anyone who writes good programs. It is pointed out that managers are usually required to evaluate the "goodness" of a programmer, yet they seem to be singularly incapable of determining the quality of a good program. We will see during this chapter that *none* of us is really in a good position to quantitatively determine the quality of a program, so a manager should not be blamed too much in this difficult situation.

Nevertheless, it seems that some programmers have a reputation in their organization for being "superprogrammers": The word will spread that Tom can turn out a large complex program in a single day, or that Alice always manages to debug her programs with a maximum of one test shot. Given this natural situation, I have occasionally amended my original question and asked my programming classes:

> Is a superprogrammer—i.e., someone who can code faster than a speeding bullet, leap over reams and reams of printout in a single bound, and generally out-program everyone else in the organization—looked upon with favor and respect by the management?

Though there is often a tremendous amount of heated debate and controversy on this point, it has been surprising how many people—especially programming supervisors and managers—have emphatically said "No!" A programmer in a large bank in Montreal put it rather well: "If my programs are outrageously inefficient, so inefficient that even my manager can tell, then I'm in trouble. Similarly, if I take ten times longer to write a program than other people in my department, I will be in trouble. For the most part, though, my manager is more interested in my ability to function as a human being in a human organization; my relationship with the computer is my own business. My manager wants me to work reasonably regular hours, interface well with other programmers and computer users, and most of all, not to cause trouble." The same management attitude seems to be quite prevalent in large insurance companies, government agencies, and banks; less prevalent in medium-sized manufacturing organizations; and generally not true in universities, research organizations, and computer manufacturing companies.

To the extent that this management attitude is true, the subject of advanced programming, and indeed this entire book, may be rather academic. What is the point of writing the most efficient program in the world if you lack other traits that make you an accepted, if not popular, member of your organization? Fortunately, the situation is generally not quite that extreme. Management *would* like to see programs with all of the "good" qualities that

we will be discussing later in this chapter; still, it is important to remember that they are often not willing to tolerate the popular image of the "computer bum" for the sake of obtaining highly efficient programs.

Note that there are different reasons for disliking the so-called "superprogrammer." Some superprogrammers can develop working programs very quickly, or can write extremely efficient programs—but they are undocumented, impossible to understand, maintain, or modify. On the other hand, there are some superprogrammers who turn out truly superlative code—and yet they are unsociable or, in the words of one manager, "a bit like Allen Ginsberg."

The most interesting point about discussing the merits of good and bad programmers is that we seem unable to measure their merits in any reasonable quantitative fashion. That is, the superprogrammer *seems* to write much better programs than his mortal colleagues—but how much better? We are at a loss in this area, for we have no way of knowing whether he is twice as good, 3.14 times as good, or ten times as good as the other programmers.

This fact was dramatically illustrated by a study made by Messrs. H. Sackman, W. J. Erickson, and E. E. Grant.[1] During a study to compare the advantages of an on-line (i.e., time-sharing) programming approach versus the standard batch programming approach,[2] a group of twelve experienced programmers were given two different programming problems to solve: One was an algebraic problem, and the other was a program that could find its way out of a "rat's maze." Careful records were kept to determine how long it took to code and debug the program. Debugging time was considered to have begun when the programmer had removed all of the "serious" compilation errors, and the debugging was considered to be finished when the program could successfully process some standard test input.

The results of the experiment are shown in Table 1.1. Note the startling ratio between the best and worst performances in the areas of coding and debugging. The ratios on all other areas of measurement, while not quite as dramatic, are still large enough to cause problems for a manager attempting to plan and schedule a programming project. As the authors pointed out in their conclusions:

> When a programmer is good,
> He is very, very good,
> But when he is bad,
> He is horrid.

[1] "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," H. Sackman, W. J. Erickson, and E. E. Grant, *Communications of the ACM*, January 1968, pages 3–11.

[2] As a point of interest, the researchers *did* find that time-sharing enabled a programmer to finish his work more quickly, though more computer time was consumed in the development process.

Table 1.1. RANGE OF INDIVIDUAL DIFFERENCES IN PROGRAMMING EXPERIMENT

Performance Measure	Worst	Best	Ratio
1. Debugging hours—algebra program	170	6	28:1
2. Debugging hours—maze problem	26	1	26:1
3. CPU sec. for program development—algebra problem	3075	370	8:1
4. CPU sec. for program development—maze problem	541	50	11:1
5. Coding hours—algebra program	111	7	16:1
6. Coding hours—maze problem	50	2	25:1
7. Program size—algebra problem	6137	1060	6:1
8. Program size—maze problem	3287	650	5:1
9. Run time (CPU sec.)—algebra problem	7.9	1.6	5:1
10. Run time (CPU sec.)—maze problem	8.0	0.6	13:1

They also pointed out that

> The observed pattern was one of substantial correlation with . . . test scores with programmer trainee class grades, but of no detectable correlation with experienced programmers' performance.

Their final conclusion was that

> This situation suggests that general programming skill may dominate early training and initial on-the-job experience, but that such skill is progressively transformed and displaced by more specialized skills with increasing experience.

What conclusions may one draw from all of this? Not much—except that we know very little about what makes programmers tick, what makes them good, or how to measure just how talented they really are. The seeming tautology that "a good programmer is one who writes good programs" is not necessarily true, according to those who hand out the raises and promotions. In addition to knowing OS and JCL on the IBM System/360, it seems that a good programmer must have a number of qualities that have nothing to do with a computer.

1.2 What Are the Qualities of a Good Program?

As I mentioned before, I often begin a course on advanced programming by asking the students for their definition of a good programmer. Not too surprisingly, I often follow this up with the question, "What are the qualities of a good computer program?" Once again, there is a variety of definitions; some of the more interesting ones are:

1. It works.
2. It works according to specifications.
3. It is flexible.