PRINCIPLES OF PROGRAM DESIGN

M. A. JACKSON

A.P.I.C. Studies in Data Processing
No. 12

PRINCIPLES OF PROGRAM DESIGN

M. A. JACKSON

ACADEMIC PRESS
LONDON NEW YORK SAN FRANCISCO
A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS INC. (LONDON) LTD.
24/28 Oval Road
London NW1

United States Edition published by
ACADEMIC PRESS INC.
111 Fifth Avenue
New York, New York 10003

Library of Congress Catalog Card Number: 75 15033
ISBN: 0 12 379050 6

Printed in Great Britain by Page Bros (Norwich) Ltd, Norwich

PREFACE

1.

This book is about programming. In particular, it is about programming for data processing applications. The main theme of the book is that programs can, and should, always be simple, even if the tasks they perform are complex. The main subject matter is a design technique which allows this simplicity to be achieved.

Traditionally, programming is distinct from system design. The system designer, usually called "systems analyst", decides what files and programs are needed in the system, and specifies these for the programmer; the programmer then writes his programs according to the designer's specifications. This traditional division of labour is absurd: it has had several damaging effects on our understanding of computer systems and on the way we build them.

First, it has helped to perpetuate the primitive idea, derived from the earliest batch-processing systems, that there is a hard boundary between the tasks of system design and program design. We expect to apply different techniques and different design criteria, and to use different tools, when the elements of our design are programs and files from those we apply and use when the elements are subroutines, machine instructions and core storage. The distinction is beginning to break down now; we recognize that JCL is a programming language (of a bizarre and unsatisfactory kind), and that the word "program" loses much of its meaning in the context of an on-line transaction processing system. The design technique discussed in this book, especially the aspects considered in Chapters 7 to 11, undermines the distinction between programs and systems in the context of batch processing also.

Second, it has obscured the nature of the systems analyst's task. He is expected to do two very different jobs: he must analyse the application needs to determine what the system ought to do if it is to serve the user well; at the same time he must design the higher levels of that system, configuring programs and files so that the work can be carried out efficiently on the computer. The first job is concerned with the techniques of inventory control, sales forecasting, production planning—whatever the particular application may be—and demands knowledge of the relevant parts of business management. The second job is concerned with computer system design, and demands knowledge of computer science. An unusually versatile person is

needed to do both of these jobs successfully. What tends to happen in practice is that the analyst concentrates on the job he likes better. Too often the needs of the user are inadequately considered; the tedious business of understanding the application and planning an ergonomically sound system is hurried through so that the pleasures of flowcharting and of laying out file and record formats can begin. Too often the resulting system specification consists of a perfunctory description of what the system will do, with a a detailed and loving account of how it will do it.

The third effect, the effect on the programmers, concerns us most directly. In many installations there is no career path in applications programming; the job is bounded above by the design work already done by the analyst, while the lower bound is being continually pushed upwards by the introduction of high-level languages, report generators and data-base software. Programmers who are technically ambitious escape into systems programming, where they can become learned in the intricacies of the manufacturer's software; programmers who are ambitious for money or position become systems analysts. Those who remain in applications programming often take refuge in an understandable, but disastrous, inclination towards complexity and ingenuity in their work. Forbidden to design anything larger than a program, they respond by making that program intricate enough to challenge their professional skill.

2.

It is already widely recognized that intricacy and complexity are programming vices; the virtues are clarity and simplicity. As we build ever larger and more powerful systems it becomes ever more important that those systems, and the components of which they are made, should be transparently simple and self-evidently correct. As Professor Dijkstra points out (Structured Programming, Academic Press, 1972):

> "If the chance of correctness of an individual component equals p, the chance of correctness of a whole program, composed of N such components, is something like
>
> $$P = p^N.$$
>
> As N will be very large, p should be very, very close to 1 if we desire P to differ significantly from zero!"

The purpose of this book is to present a coherent method and procedure for designing systems, programs and components which are transparently simple and self-evidently correct. The main emphasis is on structure—on the dissection of a problem into parts and the arrangement of those parts to form a solution.

The examples used throughout the book are necessarily small and simple;

a large and complex problem would demand too much of the available space merely to define the problem and to show the program text of the solution. The examples are therefore used to illustrate the principles of design, and the solutions given are intended to be generally valid. Little or no attention is paid to finding solutions which make best use of special facilities of a particular operating system or programming language. Above all, optimization is avoided. We follow two rules in the matter of optimization:

Rule 1. Don't do it.

Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution.

Most programmers do too much optimization, and virtually all do it too early. This book tries to act as an antidote. Of course, there are systems which must be highly optimized if they are to be economically useful, and Chapter 12 discusses some relevant techniques. But two points should always be remembered: first, optimization makes a system less reliable and harder to maintain, and therefore more expensive to build and operate; second, because optimization obscures structure it is difficult to improve the efficiency of a system which is already partly optimized.

3.

Although most of the example problems are drawn from a batch data processing environment, the design principles are applicable also to on-line systems and to scientific programming. Solutions to the problems are given mainly in schematic logic, a kind of abstract programming language which can be readily translated into any of the procedural languages in common use.

Where coding is shown, it is mostly COBOL. COBOL is still the most widely used language for data processing, and it is relatively easy for non-users to read and understand. Further, it is a very simple language, and we have restricted ourselves to an even simpler subset which is described in the appendix. So the coded solutions do not depend at all on the power of COBOL: they can be easily transcribed into ALGOL, FORTRAN, PL/I or any assembler language. They are not intended to show COBOL at its best, still less to exemplify a recommended usage; they merely show that the designs resulting from the technique can be coded without difficulty.

4.

When you read this book, you should try hard to solve each problem for yourself before reading the solution given. Most of the problems, especially in the earlier parts, can be solved in an hour or two, and sometimes in a few minutes; some of the later problems will take longer. If you have your own solution to compare with the solution given in the book you will get more

value and pleasure from what you read. Where the solutions agree, you can proceed in a warm glow of mutual approbation between writer and reader; where they disagree, you can read on in a more alert and critical frame of mind.

Exercises and questions for discussion are given at the end of almost every chapter. Each exercise is graded (a), (b), or (c). The exercises graded (a) are easy, and usually call for minor modification to a program already discussed or for practice in the use of a notation. Those graded (b) are harder, and present a non-trivial design task. Those graded (c) are harder still, and usually call for the design of a difficult program.

The questions for discussion may be of value to teachers who care to use this book, and may also give food for thought to the individual reader. Some of the questions raise topics and difficulties which are discussed later in the book.

ACKNOWLEDGEMENTS

Many of the sources of ideas for this book are already in the public domain: informed readers will recognize these, perhaps better than I can recognize them myself. I would like here to mention two sources which might otherwise go unacknowledged. First is Barry Dwyer, a colleague with whom I worked closely for several years. He provided many insights and many imaginative solutions to difficult problems, and he gave me, constantly, the benefit of his strong intellectual conscience. Second are all the people who have come to my program design courses. Their questions, criticisms and ideas have been a stimulus and an aid to refining and developing the design technique presented in this book. I never cease to be amazed that there is always something new to say about even the simplest programming problem.

CONTENTS

1. INTRODUCTION

1.1

The beginning of wisdom for a programmer is to recognize the difference between getting his program to work and getting it right. A program which does not work is undoubtedly wrong; but a program which does work is not necessarily right. It may still be wrong because it is hard to understand; or because it is hard to maintain as the problem requirements change; or because its structure is different from the structure of the problem; or because we cannot be sure that it does indeed work.

This book is about how to design structured programs so that they will be free from these faults. The basic ideas of structured programming have become widely accepted. We may summarize them briefly as follows:

> Problems should be decomposed into hierarchical structures of parts, with an accompanying dissection of the programs into corresponding structures and parts.
>
> At each level of decomposition we should limit ourselves to the use of three structural forms: concatenation (sequential flow), repetition (DO WHILE or REPEAT UNTIL) and selection (IF THEN ELSE or CASE).
>
> The GO TO statement should be avoided completely or so far as possible.

There is a brilliant description of these basic ideas, and of much more, in Professor E. W. Dijkstra's Notes on Structured Programming.

An uncomfortable analogy can be drawn between today's wide acceptance of these ideas and the acceptance by an earlier generation of programmers of the ideas of Modular Programming. The basic ideas of Modular Programming were these:

> Each program should be dissected into modules which can be separately compiled.

Modules should be as small and simple as possible within the limits dictated by the efficient use of the programming and operating systems.

Modules should be separately tested before integration into the programs which use them.

Modular Programming was not always successful in practice, for various reasons. Some compilers imposed very large overhead costs in space and time on separately compiled modules; the smallest practicable size for a module was therefore very large, and the technique useless for any but the largest problems. Some users found great difficulty in integrating modules into workable programs; during "integration testing" many interfaces between modules had to be respecified and many modules rewritten, at a cost greater than the savings achieved in originally constructing the modules. Certain promised benefits were obtained only rarely: few users managed to create a library of general-purpose modules and so reduce the amount of new code to be written for each successive project; many users found that program maintenance became harder, not easier, because many modules had to be amended and recompiled where previously only one monolithic program was affected. Almost all users became conscious that they had a major new problem in program design: what was the best way to dissect a program into modules? —or, more succinctly, what is a module?

This last problem was crucial. But there were no good answers to the questions. Some answers were useful for a limited range of simple problems: "the program should have a main-line control module with subordinate modules to process transactions". But for the most part the answers permuted a standard range of buzz-words—"logical entity", "functional integrity", "generalized logical function" and many others—and no-one was any wiser for them. Programmers who had previously written good monolithic programs now wrote good modular programs; programmers who had previously written bad monolithic programs now wrote bad modular programs.

1.2

We face a similar difficulty in Structured Programming. It is not enough to decide that a program should be built of DO WHILE and IF THEN ELSE constructs: the crucial problem is to decide what particular DO WHILE and IF THEN ELSE constructs are needed for this particular program, and how they should be fitted together. If the structure is wrongly designed we will not be saved by the fact that each individual part is well formed.

As an illustration, consider the following trivial problem.

PROBLEM 1—MULTIPLICATION TABLE

A multiplication table is to be generated and printed. The required output is:

```
1
2     4
3     6     9
4     8     12    16
5     10    15    20    25
...   ...   ...   ...
10    20    30    40    50    60    70    80    90    100
```

The table is to be printed on a line printer, using the statement DISPLAY PRINT-LINE to print each line as it is generated.

Here is a very badly designed program to solve this problem.

```
...
DATA DIVISION
WORKING-STORAGE SECTION
77  LINE-NO PIC 99.
77  COL-NO PIC 99.
01  PRINT-LINE.
  02  NUM OCCURS 10 PIC ZZZ9.
PROCEDURE DIVISION.
PSTART.
    MOVE SPACES TO PRINT-LINE.
    MOVE 1 TO LINE-NO.
    MOVE 1 TO NUM (1).
    PERFORM PLINE UNTIL LINE-NO = 10.
    DISPLAY PRINT-LINE.
    STOP RUN
PLINE.
    ADD 1 TO LINE-NO.
    MOVE 0 TO COL-NO.
    DISPLAY PRINT-LINE.
    PERFORM PNUM UNTIL LINE-NO = COL-NO.
PNUM.
    ADD 1 TO COL-NO.
    MULTIPLY LINE-NO BY COL-NO GIVING NUM (COL-NO).
```

The program was designed by drawing a flowchart, and coded from the flowchart. It works correctly, producing the required output. The coding itself is well-formed: the PERFORM statements are correctly coded repetitions and the rest of the logic is sequential flow with no GO TO statements. And yet the structure is hideously wrong.

Consider what changes we would need to make to the program if the problem were changed in any of the following ways:

print the upper-right triangular half of the table instead of the lower-left triangular half; that is, print:

1	2	3	4	5	6	7	8	9	10
	4	6	8	10	12	14	16	18	20
		9	12	15	18	21	24	27	30
							...	... ...	...
								81	90
									100

print the lower-left triangular half of the table, but upside down; that is, with the multiples of 10 on the first line and 1 on the last line.

print the right-hand continuation of the complete table; that is, print

11	12	13	14	15	17	18	19	20
22	24	26	...	...	...	...	...	...	40
...	...	...	...	...	...	...	...	...	...
110	120	130	140	150	...	...	190	200

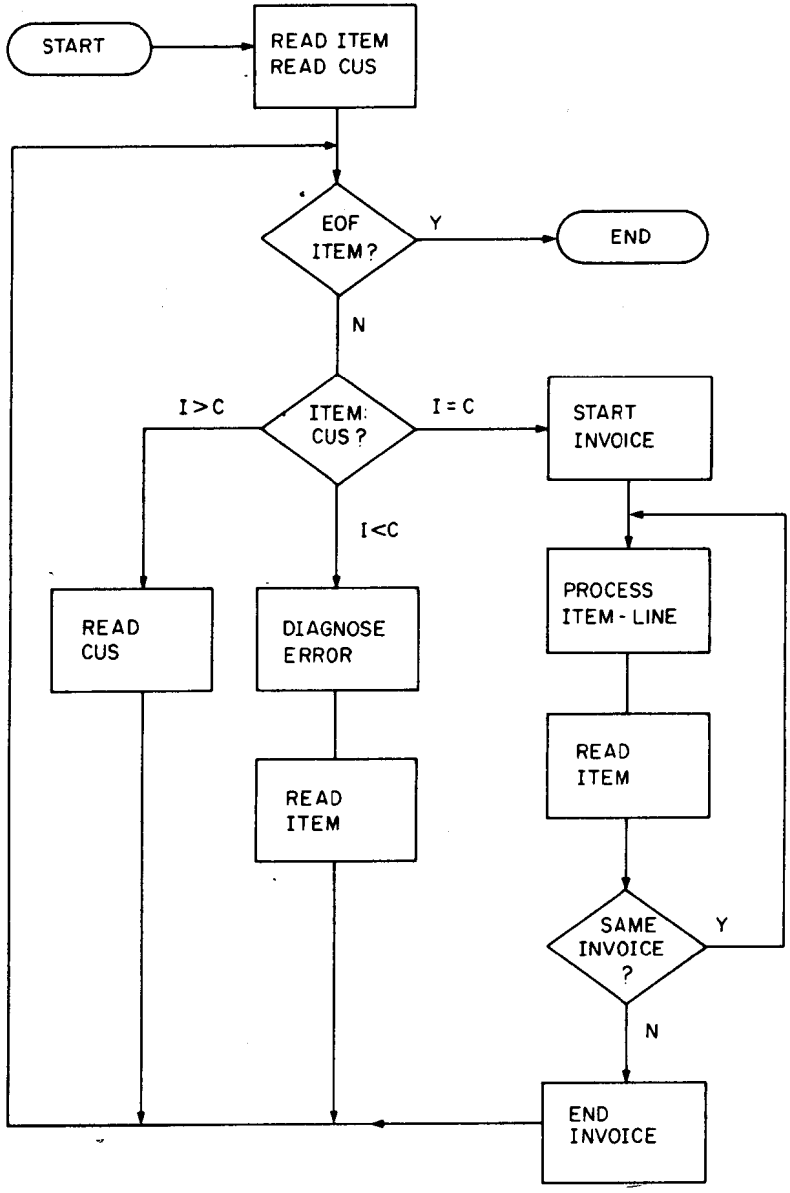
All of these changes are awkward—or as awkward as changes can be when the problem is so trivial and the program so small. The first change affects only the choice within each line of which numbers are to be printed and which omitted; instead of beginning at NUM (1) and continuing to print up to and including NUM (LINE-NO), we want to begin with NUM (LINE-NO) and continue up to an including NUM (10). We ought to be able to make a localized change to the program—perhaps to the second and fourth statements of PLINE—but we cannot. The changes needed in the program amount almost to a complete rewriting. We are defeated similarly by the second and third changes.

The essence of the difficulty is this. We wanted to make simple and localized changes to the specification: to alter the choice of numbers to be printed within each line; to alter the order of printing the lines; to alter the choice and values of numbers to be printed in each line. We therefore looked to make similarly localized changes to the program: where is the component which determines the choice of numbers to be printed? where is the component which determines the order of the lines? where is the component which determines the values of the numbers? And the answers were not so simple as we hoped. PLINE appears superficially to be the component which processes each line. In fact, however, PLINE prints line N and generates line N+1 when it is executed for the Nth time. So PLINE is executed only 9 times, and the first line is generated by PSTART and the 10th line is printed by PSTART. Furthermore, the printable values in each line persist in the next line, unless they are overwritten; PRINT-LINE is cleared only once, at the beginning of PSTART. So in considering what is to be printed in each line we have to bear in mind what was in the previous line.

In short, the program structure does not match the problem structure. The program should instead have been as follows:

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
77  LINE-NO PIC 99.
77  COL-NO PIC 99.
01  PRINT-LINE.
  02  NUM OCCURS 10 PIC ZZZ9.
PROCEDURE DIVISION.
PTABLE.
    PERFORM PLINE VARYING LINE-NO
                FROM 1 BY 1 UNTIL LINE-NO > 10.
    STOP RUN.
PLINE.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PNUM VARYING COL-NO
                FROM 1 BY 1 UNTIL COL-NO > LINE-NO.
    DISPLAY PRINT-LINE.
PNUM.
    MULTIPLY LINE-NO BY COL-NO GIVING NUM (COL-NO).
```

The paragraph PTABLE processes the whole table. The paragraph PLINE processes each line. The paragraph PNUM processes each number. The table


START
READ ITEM
READ CUS
EOF ITEM?
Y
END
N
I>C
ITEM: CUS ?
I=C
START INVOICE
I<C
READ CUS
DIAGNOSE ERROR
PROCESS ITEM-LINE
READ ITEM
READ ITEM
SAME INVOICE ?
Y
N
END INVOICE

consists of 10 lines, and PTABLE executes PLINE 10 times. Each line consists of LINE-NO numbers, and PLINE executes PNUM LINE-NO times. There is a perfect correspondence between the program structure and the structure of the problem.

1.3

Here is another illustration of the difference between right and wrong in program structures.

PROBLEM 2—PRINTING INVOICES

A serial master file contains customer name and address records, arranged in ascending sequence by customer number. Another serial file contains billable item records, arranged in ascending sequence by date within invoice number within customer number.

These two files are to be used to print invoices. There may be more than one invoice for a customer, but some customers will have no invoices. Due to punching errors, there may be billable item records for which no customer record exists; these are to be listed on a diagnostic file of messages.

Shown opposite is a flowchart of a solution. We assume that at the end of each file the associated record area is filled with artificially high values.

Here, in abbreviated and informal style, is skeleton coding for a COBOL program corresponding to the flowchart. . . .

```
...
PROCEDURE DIVISION.
PSTART.
    Read item file.
    Read customer file.
    PERFORM PROCESS-ITEM UNTIL end of item file.
    STOP.
PROCESS-ITEM.
    IF CUSNO IN ITEM-RECORD > CUSNO IN CUS-
                                              RECORD
    Read customer file
    ELSE IF CUSNO IN ITEM-RECORD < CUSNO IN CUS-
                                              RECORD
```

```
        Diagnose error
        Read item file
        ELSE PERFORM PROCESS-MATCH.
    PROCESS-MATCH.
        PERFORM START-INVOICE.
        PERFORM PROCESS-ITEM-LINE-AND-REC
                                UNTIL end of item file or new invoice.
        PERFORM END-INVOICE.
    PROCESS-ITEM-LINE-AND-REC.
        Process item record, producing invoice line.
        Read item file.
```

Once again, the program works, and the coding is impeccable. But the structure is utterly wrong.

Again, we can see how wrong it is by considering some likely changes to the problem specification. This time, the changes are to be applied cumulatively:

> Print on the diagnostic listing the customer numbers of those customers for whom at least one invoice has been produced
>
> Print on the diagnostic listing, and mark with an asterisk, the customer numbers of those customers for whom no invoice has been produced
>
> To each customer number printed by the first change append the total amount invoiced for that customer
>
> Instead of diagnosing each item record in error, print only the customer number, marked with an "E", for each number for which at least one error item exists.

None of these changes is impossibly difficult. Anyone who has worked in data processing has seen programs changed in this kind of way, and probably successfully changed. But the changes are much more difficult than they ought to be, and as we make each successive change the conviction grows that we are storing up further difficulties for ourselves. Sometimes, after many changes, the program becomes so hard to understand that any further change is dangerous, and a complete redesign is then necessary.

What makes the changes so difficult? They all call for the insertion of coding into the program to process one customer. For the first change we need to insert a statement "print customer number" at a place in the program where it will be executed once for each customer who has at least one invoice. For the second change we need to insert a similar statement at a place