SOFTWARE DESIGN FOR MICROCOMPUTERS

CAROL ANNE OGDIN

SOFTWARE DESIGN FOR MICROCOMPUTERS

CAROL ANNE OGDIN

Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

Library of Congress Cataloging in Publication Data

OGDIN, CAROL ANN, (*date*)
Software design for microcomputers.

Includes index.
1. Microcomputers—Programming. I. Title.
QA76.6038 001.6'42 78-5801
ISBN 0-13-821744-0
ISBN 0-13-821801-3 pbk.

EC82/01

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *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*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Preface

"Software Design for Microcomputers" distills much of the recent experience gained by trial-and-error in the software industry. From this sorting of the best of available techniques, I hope that you, the reader, will be spared some of the same mistakes others have made. I made the assumption that you are about to embark on a software design task (probably on a micro, but the principles work for any common digital computer), and that you want to do the best job possible. I have tried to select the best methods from all of the available literature and meld them into a cohesive methodology that experienced digital designers and engineers can apply with little fear of failure.

There are many books on programming—but few on the subject of software design. And, so far as I know, none of the available books assume that you are an intelligent, rational designer with some experience that serves as a valuable starting place. Like its companion, MICROCOMPUTER DESIGN, this book has already been "field-tested." About two-thirds of this manuscript originally appeared in *EDN* magazine's June 5, 1977, issue under the title "Software Design Course." In recognition, that issue was accorded the American Business Press' Jesse Neale Award as the best contributed series of 1977.

The purpose of SOFTWARE DESIGN is not merely to reprint a series of articles, but to add to the basis of knowledge you already have. You may be new to software, having learned the rudiments while reading MICROCOMPUTER DESIGN, or you may be experienced in digital design with a smattering of programming background, or you may have last brushed against a computer in a FORTRAN course in college. No matter what background you bring to it, "Software Design" will show you a step-by-step approach to the design, implementation

and checkout of reliable and valuable software. The book is arranged in a progression of thirteen chapters, from fundamentals review to sophisticated detail.

Chapter 1 reviews the basis for software and explodes some myths. Software design is just like hardware design—no more difficult, no easier. The same design disciplines that are applied to digital systems in TTL or CMOS can be applied to programs written for a computer. The emphasis for the entire book is established with the twin premises that: 1. Literate man of the late 20th Century must know how to program; and 2. Programming is a cerebral exercise conducted by intelligent people.

Chapter 2 describes in detail the basic steps in successful software design. By intentionally drawing the distinction between designing, programming, and coding, we separate a formerly confused morass into a logical sequence. Then we show the economic consequences of doing a poor design job and expecting to recover during the debugging phase of the project.

Chapter 3 begins "Software Design's" radical departure from most introductory programming texts. Instead of showing how to write code, this chapter shows how to design data structures. *This* is one of the major keys to successful software. A set of possible variants on data structuring are shown, and some empirical rules for describing all possible data structures are outlined.

In Chapter 4, the logical "dual" of data structures—procedures—are described in equivalent detail. Parallels are constructed between data structures and procedure structures, and the power of switching from one to the other during design is emphasized.

In Chapter 5 we begin to apply these techniques to the requirements that are common to both hardware and software. By recording each step, from concept of need to final algorithm, this chapter shows how experienced designers work.

Alternative ways to describe the algorithms for implementation as a computer program are shown in Chapter 6. First, flowcharts are debunked. Next, several alternative ways of describing software during the design process are presented. Choices among these different documentation and design tools are also explained.

Chapter 7 may seem out of place in a book on software design, yet it may be the most important part of the entire work. Documentation is done *before* a program is written, not after. Chapter 7 shows a simple way to document software (and hardware) designs in a way that encourages revisions during the design stages and discourages them after the design has been settled.

Chapter 8 introduces one of the important parts of programming: coding. We have elected to use the BASIC language only because it is so easy to read. Although a powerful language, it is not the only tool in a programmer's kit. The rudimentary statements in the BASIC language are revealed in sufficient detail for subsequent chapters to show actual program implementations.

Coding a properly designed program into BASIC and onto the computer is the subject of Chapter 9. The steps, incorporating the concepts of Chapters 2 through 8, are outlined in a series of illustrations, each one a more elaborate design of the one preceding. Finally, an actual computer print-out shows all the actions taken in

the testing and debugging of a simple program. Coincidentally, the application selected is unusual: the plotting of an arbitrary function on the printer or terminal of the computer.

Chapter 10 deals with the tricks and techniques of the inexact art of testing and debugging computer programs. Like design, testing is a mental activity. Most practical programs cannot be tested exhaustively, so some intelligence must be applied by the test-case designer to inspire confidence in the resulting software product.

Chapter 11 deals with several other popular languages. The discussion shows the extensions provided in advanced versions of BASIC, then illustrates programs in FORTRAN, COBOL, APL, PASCAL and Assembly Language. In each case, the advantages and disadvantages of each language are provided so you can select the best language for your own programming needs.

The special requirements of real-time programs on minis and micros are treated in detail in Chapter 12. Since most microprocessor applications involve real-time, and real-time is the bane of programming reliability, compromises are essential. Chapter 12 shows how to judge those compromises and how to design highly reliable real-time software.

Systems software, programs that aid program development, are discussed in the final chapter of the book. Monitors, supervisors and operating systems are described, as well as some of the considerations in their selection and use. Finally, principles behind language translators and program loaders are discussed, so you can understand manufacturer's software literature.

These chapters describe techniques I use every day and have taught to many other people. If you follow them carefully, you will be able to produce reliable, useful and valuable programs. Furthermore, you will have a distinct advantage over most "experienced" programmers who resist learning these new techniques because they believe they know all there is to ever know about programming, designing and coding. I do not believe that I will ever know it all—but these few techniques keep me out of the most serious kinds of troubles to which programmers seem heir.

CAROL ANNE OGDIN

Contents

1

What Is Software?

Software is no longer an avoidable subject. Although computers have been accessible to engineers and system designers for over 30 years, many of us have avoided learning the new skills required. And, of course, many of the computer experts have made a simple topic complex. Actually, as you will discover in this design course, software is not at all difficult to learn. Much of the knowledge you already have can be redirected toward the new implications of computer programs; learning how to program need not be a painful experience.

This design course will be an unconventional treatment of software. You will not find another boring introduction to binary arithmetic; if you have ever designed a digital circuit you know that already. You will not find another basic introduction to elementary computers; as an intelligent, well-read designer, you obviously already have these ideas firmly grasped. In this design course we are going to concentrate on the practical steps you must take in order to understand, use, and create software in today's electronic environment.

What Is Software?

Software is more than a collection of holes punched in cards or paper tape. It includes the whole gamut of nonelectronic support to computers. When we talk of software, we mean the computer programs (in all their various forms), the instructions for use and the user's manuals, and the necessary design documentation.

If we look at the computer itself, we can see it, feel it, even squeeze it. This part of the system is called the *hardware.* But a general-purpose computer is a use-

less lump of electronics until and unless it is provided with a program in the computer's store. The physical storage medium is part of the hardware; the particular binary state of each individual storage element is not something we can generally feel, see, or squeeze. This particular combination of binary states contains the program (and probably some data). Since we cannot physically handle it, we call it *software.*

There are three different kinds of software:

1. Applications programs. Software that is especially written to solve some particular problem.
2. Systems programs. Software that usually comes with the computer from the manufacturer and is designed to make the creation of applications programs easier.
3. Documentation. Software that shows us how to use the programs and how to modify them for special needs.

Sometimes, only the second kind of software (system programs) are considered to be "software." According to accepted industry standards, however, if you take

The Gutenburg Era

To be a good programmer today is as much a privilege as it was to be a literate man in the sixteenth century.

Andrei Ershov

Microprocessors have brought us all into the Gutenburg era of computing. It is almost as inexpensive to reproduce and distribute a *process* as it is to reproduce and distribute *information.* Just as Gutenburg's invention allowed the inexpensive reproduction and distribution of information in the form of printed and bound books, the microcomputer will have an astounding impact on society in the next 25 years.

The effect on electronic designers will be no less astounding. No longer need we concern ourselves with the management of vast armies of monks, secretly enscribing illuminated manuscripts of COBOL and FORTRAN, each with their own individualities and idiosyncracies. Instead, we can look forward to the creation of an elite cadre of software authors who create the major works and to the eventual acquisition of programming skills by the general populace.

Literate men of the late twentieth century must know how to program.

away the physical computer from a system, what you have left is called the software.

You will notice an emphasis on including documentation in the definition of software. That is intentional. Throughout this design course you will find a recurring theme of documentation. Good programs are characterized by good documentation. Furthermore, good documentation is written *before* the program itself is written. All too often when we think of software we imagine only the computer program as it resides in the computer's storage medium. That is the hallmark of the novice. The experienced computer user looks to the documentation, just as the experienced electronics designer looks beyond the schematic to the supporting documents. A good design document completely explains what the program does. The program is a formal restatement of the design in binary digits that the computer can understand.

Software Environments

Software exists in relation to some computer (or computers). It is important to note the differences among kinds of computers and the various environments in which software exists. The concepts of programming and software design are the same, regardless of the software's environment. However, you cannot effectively design without having some awareness of the various environments you may be called upon to exploit.

Certainly, today's designer is faced with the novel opportunity of the microprocessor and microcomputer. These devices provide the greatest impetus to learning about software only because their application domain is so pervasive. To ignore software (and therefore to ignore micros) is as foolhardy as trying to ignore integrated circuits might have been a few years ago, or transistors a decade before that. Micros represent an essentially "raw" computing capacity that is unadorned by much additional software to make using them easy. However, if history repeats itself, more and more systems programs will become available, making the use of micros easier and easier. That, in turn, will drive costs down even more and will encourage more people to use them. With more users, there will be more incentive for the development of more and better systems programs that will, of course, encourage even more use of micros.

The history of the micro is already paralleling the recent history of the minicomputer, which was originally introduced as a relatively low-cost system component (priced around $25,000; then, computers cost over $250,000). The mini was also without much software. However, today's minicomputers are augmented with a wide range of software alternatives from the simplest (not unlike the common microcomputer support available) to the most complex. Some minicomputers even have whole systems programs called *operating systems* that allow multiple simultaneous users of the same computer.

The large-scale computers of today are generally used to handle problems that require access to large files of data or massive computation. The demise of the large-scale computer has been predicted for years, but it is unlikely. The centralization of data files for an organization requires a single computer complex. And many of the problems that require high-speed computational ability need that capability for only a few seconds each day. A large centralized computer allows many, many users who have similar kinds of problems to share a single resource.

There are some subtle details in the design of software for these different environments, but the techniques are almost the same throughout. For example, if you write a program for a microcomputer, you will have to descend into a morass of details to handle the individual input/output circuits. On a minicomputer you will be able to avoid some of this detail by using some of the features of the systems programs provided by the manufacturer. On a large-scale computer you will probably be prevented from being able to get involved in the details of input/output programming. At the microcomputer end of the spectrum there is more "raw" input/output capability, but at the large-scale computer end you will find the process of implementing a program significantly easier.

The environment in which software gets written also includes other software. When you write a program, you will be using other programs to help you do your work. Instead of having to write programs in the one's and zero's of binary, you can write them in a much more readable form; a system program is used to "translate" your readable form into the computer's required binary form. In fact, as you begin to adopt the use of a large-scale computer (or, to a lesser extent, the minicomputer) you will find that you will spend less time understanding the underlying computer and more time learning about the systems programs provided as the software environment of that computer system. In this design course you will learn how to understand all of the system's software.

Uses of Computers

If you look around you will find a wealth of opportunities to exploit computer technology in the electronics industry. In this design course we will stay away from the traditional management-oriented and administrative uses of computers. Our examples will be drawn mostly from the applications for microcomputers and minicomputers with which you will most likely become involved. However, there are numerous other uses you might discover, even if you don't design around micros and minis.

For example, have you ever had to prepare a proposal for a large project? (There are two kinds of engineers: those who have and those who will.) How do you price out different alternatives for a project with lots of people and only a certain amount of time? How do you try out various alternatives, for example, making certain subsystems and buying others? To try out all of the alternatives by using a desk calculator can be terribly time-consuming. This is a natural applica-

tion for a computer. You can use the organization's data processing department's computer, or an available minicomputer, or even a terminal to a remote time-sharing system. You can design and write a small program that does all of the routine math and prints out an annotated result; then, by changing the data you can produce different reports that can be compared for effectiveness.

Prewritten (*canned*) programs can be leased or bought. These programs can be used for project scheduling and analog circuit design and for solving simultaneous equations and so on. You don't necessarily have to be able to program in order to buy and use these programs, but you do have to know enough about software to be able to evaluate what you are buying.

There are even more novel applications for your new-found software skill. Have you ever needed a generator of complex digital or analog signal sequences? The experienced computer user plucks a common computer off the shelf, programs it to suit, and generates an easily changed signal sequence. It is easiest to visualize if your application is digital, but computers can also be used to synthesize analog signals. And a derivative application is the use of a computer to simulate some unavailable piece of more complex equipment. If you are designing a subsystem of an aircraft instrument panel, you can actually create all of the complex interacting signals that arrive from the various sensors to simulate an actual flight. This can be used in system testing and for important demonstrations.

The list of potential computer applications that are open to the knowledgeable user of software is virtually endless. But the most important use of your software skill is in the planning of your own career. The designer of good software is a recognized asset in almost any organization. Because of your new abilities, you may find yourself given more responsibility and presented with more interesting technical challenges. And in today's world, more and more problems are going to require the systems approach that is inherent in good software design; learning software has more to offer than just a new bag of tricks. It may be a significant step in your professional advancement.

What Is Programming?

Contrary to all the popular books on the subject, programming is *not* the writing of cryptic statements in FORTRAN, BASIC, or COBOL. Programming is the act of designing a specialized sequence of instructions for a fast and faithful clerk to carry out endlessly (and mindlessly). If you imagine a computer as an unerring clerk to whom you must provide detailed and complete instructions to carry out some task, you will begin to appreciate what has to be done in the authorship of a program. At the very minimum, you have to know more about the intended application than you would have to know if you were to do all the work yourself. The computer cannot fill in its own gaps in knowledge; it can only follow the steps in the program.

The first step in programming a computer is to understand what has to be done. It is at this point that most computer program designs fail. A clear and complete description of the intent and requirements of the software must be documented before any design is attempted. If you design any system, whether hardware or software, without understanding the needs, you are doomed to having to tear it all down and redesign over and over again. After we have introduced some of the basic concepts you need to know, we will show you how to document the requirements in a clear and consistent way.

The second step in programming is to design the software system that will be required. This generally starts by breaking the requirements down into groups, each individual group representing a semi-independent module. Each module is further broken down into individual units until a level of detail is reached at which each unit can be thoroughly understood and completely designed. These first two steps demand a significant discipline to rigorously document each important decision.

The third step in programming is to implement the program. The design is manually translated from the documented form into a form that is acceptable to the computer; this form is called a *programming language.* This phase is called *coding,* and it is the phase at which most people (erroneously) begin.

The next step in programming is generally semiautomatic and is called *translation.* Your program (in an appropriate language) is translated into an equivalent binary form that the computer will be able to execute. This translation is done under the aegis of a systems program called an *assembler* or a *compiler;* any errors you may have made in writing in the programming language are detected and reported to you by this systems software.

After you have an understood, designed, coded, and translated program, you must test it. After finding one or more faults, you must debug it to find what has to be changed in order to make it function correctly. Testing is an art of its own, as is debugging. Chapter 10 is devoted to these two topics. Testing and debugging are generally done repeatedly until the program achieves a satisfactory level of quality of behavior. The objective of testing is to find the errors that may have been made in the entire process; it should not be the point at which you attempt to make up for your lack of original understanding of the problem that you are trying to solve.

All too often neophyte programmers see the actual coded program as the only necessary object of the programming exercise. Experienced programmers know that there are many preliminary steps that must be taken. The experience level of programmers can be judged by finding out how soon they begin writing code for the computer: The novice begins almost immediately and assumes this is the right thing to do. The more experienced programmer does a lot of preliminary paper work and produces a series of design documents; actual coding doesn't begin until much later. In general, the earlier the coding is done the poorer it will be because at this stage the problem is not usually well understood.

The novice programmer typically grasps any "corner" of the intended application (often the one he "understands" best) and begins to chop away at it. If there is to be a terminal, he may start to create part of a program to read in data from that terminal. It doesn't occur to him that he doesn't know what to do with that data once it is in; it doesn't occur to him that this may not be the right way to cope with the terminal.

The experienced programmer knows that quickly rushing off to write code will produce useless software. He first sets out to understand the problem, designs the software, specifies the steps required in the solution, and finally establishes the requirements of each module and how they are to be fitted together. Eventually, he will achieve enough of an understanding of the entire system (and its partial evolution at some stage) to be able to specify the characteristics of the various components of the entire system, such as the terminal and its associated software.

Growing Complexity of Problems

All of the easy jobs have been done. From now on the applications for electronics (and for computers) will be more difficult and more challenging. The simple, sweeping solution will become more elusive and less likely to be applicable.

Engineers have tended to address problems that were easy to solve, for example, the design of radio transmitters and receivers. More difficult problems, for example, air traffic control (that may use radio transmitters and receivers as subsystems), are being tackled now. The really difficult problems, for example, designing a total transportation system from portal-to-portal, are much, much more difficult to solve and probably represent the challenges of the next 25 years.

The more complex problems do not have simple, fixed solutions. Solutions may have to be able to adapt to changing conditions, and this adaptability is most easily provided in the form of software for a computer. The earliest traffic lights, for example, had simple sequential timer control of an intersection. Later, as traffic grew, multistate controllers were created that could operate on different cycles, depending on the time of day and the instantaneous traffic. Now, with microcomputers at each corner, the traffic signal's sequencing may be the result of adaptation to the overall traffic density and flow. In the future all of the microcomputers may be linked together into a complex network that automatically optimizes the sequences of traffic lights in order to maximize the efficiency of the traffic system as a whole. Since nobody knows how to do this last step yet, it is the challenge of the future.

Coding may take a few hours to complete and debug and there will be little likelihood that it will have to be torn down and written over again as a result of design changes.

The novice is working upward from the most detailed available level of knowledge toward the eventual system. The experienced designer is working from system specifications downward toward the actual program implementation. In the novice's case, the quickly written program is produced with the assumption that it will ultimately prove useful to the whole design. This is—at best—a risky prediction. The experienced designer is betting that at some point the required functions can somehow be implemented; this is most certainly a surer kind of prediction.

Programming is just as difficult (and as easy) to do as electronics circuit design. Writing code down on paper is just a bit easier than wrapping wire or soldering, that's all. The same design and debugging steps have to be taken in either case. If you can habitually read the specs for a project and then go directly to the bench and wire up some circuitry that works, you are one of those rare people who can write correct program code "on-the-fly." If, however, you are a mere mortal like the rest of us and have to spend some time consulting books and trying different alternatives on paper before you can produce a schematic, you will find that the same methods also work best with software.

Some engineering-dominated companies relegate programmers to support and technician-level roles. There organizations are easily identified. Their programmers spend time producing programs that solve "flaps," are transient instead of steady-state, are devoid of documentation, can only be used by the programmer who originally wrote them, and are not saved as a part of the organization's software resources library. When the programmer leaves the group, any attempt to use the program again or to extend it results in paying another programmer to rewrite it, often with the same undesirable side effects. Experienced programmers know that there are no "quick-and-dirty" programs—not quick, anyway.

The objective of the rest of this design course is to show you how to specify, design, implement, and test reliable and efficient software with the least amount of fuss and bother. If we are both successful in our communication, you will learn how to avoid the pitfalls that have trapped so many budding programmers in the past. Even if you never program yourself, having the skill in reserve is a worthwhile objective. You never know when you will have to communicate with another programmer.

2

Procedures and Data

In its raw, delivered state, just after power is applied, a computer is a pathetic piece of universal electronics. It is raw potential, but it has no purpose. A computer, as you have already learned, is capable of doing only two things:

1. Fetching an instruction from some storage medium
2. Executing that instruction

But it does so endlessly and rapidly. What we lack in our raw computer is a *program,* the group of instructions that will be sequentially executed to achieve a specific and defined objective. The combination of a raw, unadorned computer with a well-written and debugged program forms a special-purpose machine. What you do with it depends on how you program it.

Analogy of a Program

We have all had the exasperating experience of having to deal with a faithful, loyal, and dedicated employee who follows every instruction precisely as given but who uses no common sense. Well, conjure up a person like that and let him be our computer for a moment. We will supply the faithful clerk with a calculator (an arithmetic unit) and a sheet of instructions (Figure 2.1). If our clerk is unerring in following the instructions, the OUT basket will end up with a stack of papers that is a transformed version of the original IN basket contents. If the desired output from the process is the correct transformation of the input according to the instruction sheet, we have a correct and valid program.

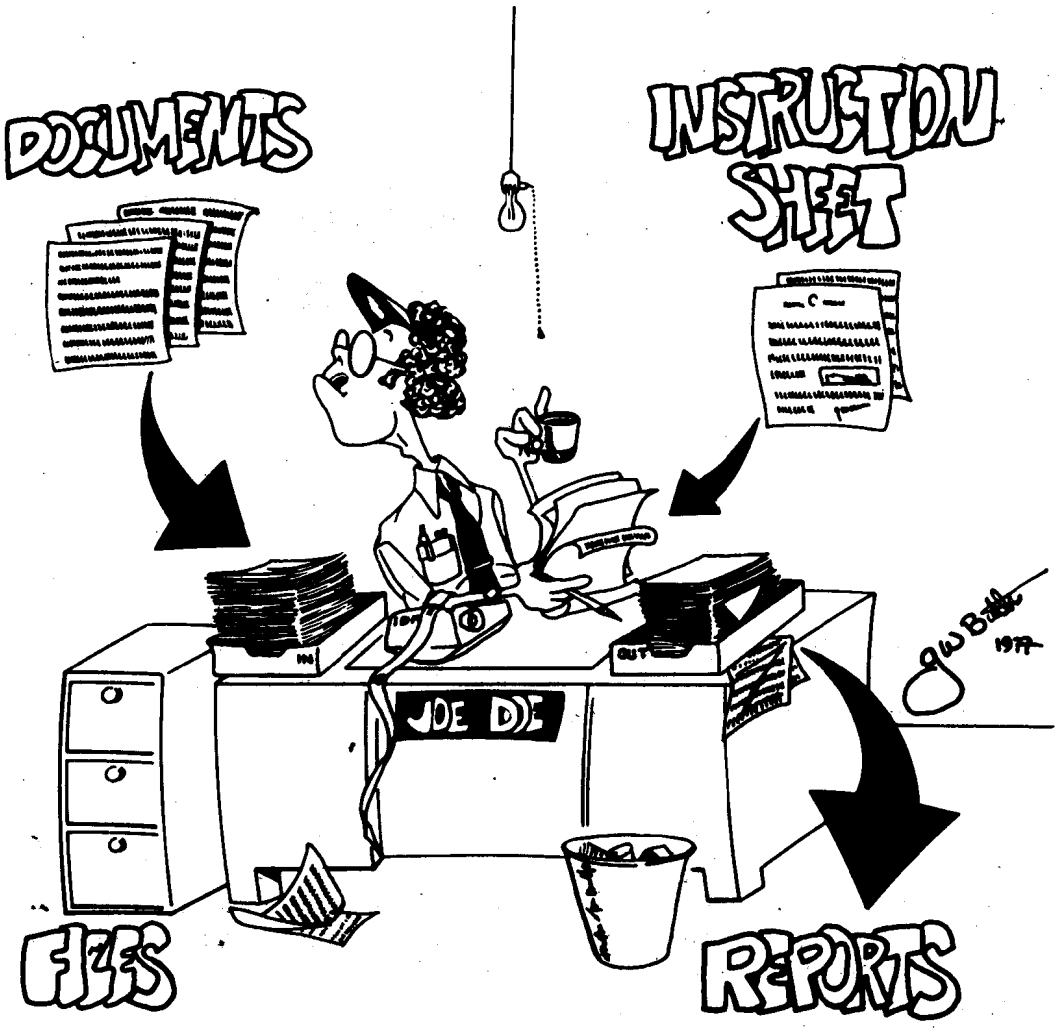

Figure 2.1 A devoted clerk can be conceived of as an analog of a digital computer; it is also a way to test your instructions.

Such a program cannot be supplied to a computer, of course. Computers are not very good at reading the English language. In Figure 2.2 you can find three distinct meanings for the sentence. (What does a computer do in that case?). Furthermore, we have not handled all of the cases for even this simple condition. For example, what if the OUT basket becomes full and overflows onto the floor? What should we do on our instruction sheet to tell the clerk to watch out for overflow and do something different? Once your computer program is underway doing useful work, you won't be able to intervene and handle all of the special cases like the overflow of the OUT basket.