IT2205: Programming I

Section 4

Computer program design (05 hrs)

Structured Design and Interactive Programs

INTRODUCTION

We have, by now, looked at the major aspects of program design. We can implement the major elements of a design, and we are familiar with two of the major formal design tools. It is time to take a step back and place our knowledge within a broader framework. What makes one design “good” and another “bad” (or at least “less good”)? How do we recognize a “good” design? How can we define a “good” design so that others can recognize one? It is not enough to say, “I know it when I see it.” We must be able to define our design goals and requirements.

STRUCTURED DESIGN

One of the terms we will use here is “structured” design. What are we adding by the term “structured”? Over the past 25 years, a form of program design has been developed that emphasizes a certain methodology of design specific requirements for the completed design. Designs that meet those requirements are said to be structured. Designs that followed the earlier process of design are said to be unstructured. We will look at the methodology and requirements of structured design.

REQUIREMENTS OF STRUCTURED DESIGN

TOP-DOWN DESIGN-We have looked at our design problems so far by asking first what are the major tasks, then breaking those tasks down into subtasks, and continuing that process until we had complete list of *what* needed to be done. The key point is that we started with a broad view of the design and continued to break that broad view down into smaller and smaller sections until we were down to the level of the smallest detail needed. This process is Top-Down Design. It is also known as Step-Wise Refinement and Functional Decomposition. However, top-down design goes further than just creating our task list. We also write our formal design in the same manner, starting with the broad view (our Mainline module) and working down to the small detail modules.

Our completed design has the same “look” as our original task list because we present it from major tasks broken down to minor tasks. We continue in the same manner to code and implement our design. Rather than cade the entire program and try to run it all at once, we code it in sections (our Mainline module, then add the input and output modules, then add the calculation modules, etc.). We continue to use the top-down philosophy with our program documentation. We begin to create our documentation package when we begin to list our tasks. The completed packages will show the completed design, the source code, the results of test runs, and any notes accumulated about the program.

LIMITED TO THE THREE MAIN STRUCTURES-We use the sequence, selection, and loop structures, as well as the few accepted variations of those structures (UNTIL loop as a variation of the WHILE loop, case structure as a variation of the nested IF). You are probably asking, “What else is there?” As you will see later in this chapter, it is possible to create designs (and programs) that ignore the requirements we have set up for the three structures. The designs may even work, but they will almost always be difficult to follow and therefore difficult to modify.

PROGRAMMER TEAMS-Major programs should be created by programmer teams working cooperatively under a team leader. Programming teams provide several advantages to the designers. Instead of one designer working months or years on one program, a team can complete the program in much less time. A single programmer has no one who is already familiar with the program to help resolve particular problems. A team will be in regular consultation and will be able to put their heads together quickly and efficiently to solve design/programming problems.

STRUCTURED WALK-THROUGHS-A structured walk-through, a formal review of the design (and perhaps code) by a review team of peers, should be used with major programs. The goal is to identify potential team of peers, should be used with major programs. The goal is to identify potential problems before the design is cast in concrete. The walk through team should be given copies of the design in advance and “walked through” the design by the primary designer. The goal is to find the potential errors, not to fix them. Identified problems are referred back to the design/programming team for revision.

MODULAR CONSTRUCTION-Designs are written using a modular construction. Each module is dedicated to a single function (or at times to a particular set of lesser functions grouped because they occur at the same time). Within a module, statements should be tightly related with a clear and obvious reason for the inclusion of each statement. Very loose relationship should exist between modules. We have looked at modules before, but the use of programming teams makes modularity much important. One programmer may write one module, and other programmer (perhaps working at a different site) may write another module, and the two modules may need to share data. The way on which a program is broken into modules may make it easier (or more difficult) for the two programmers to coordinate their work. The structured walk-through is where the team makes sure the modules fit together.

SINGLE ENTRANCE-SINGLE EXIT-Every structure, every module, and every program should be limited to a single entrance and a single exit. This is really a restatement of earlier rules, but it is important to emphasize. Early designs were often made to work by one-way branching. When we call a module, the return form the module is automatic. It is a round trip ticket since when the modules is completed, control automatically returns to the statement following the call. Earlier designs used a “GO TO” that branched to another location without any return implied. The GO TO was often used for multiple exits from IF statements and from loops rather than controlling for data variations within IF or loop.

FLAGS, SWITCHES, AND Ifs-The use of flags, switches, and IF statements make the single entrance/single exit rule possible.

GOALS OF STRUCTURED DESIGN

As we will see when we look at some of the history of structured design, the rules given above were developed in direct response to the then current problems in the computer program development world. The goals essentially were to correct those problems. The goals of structured design continue to be practical, production-oriented goals.

LESS TESTING/DEBUGGING TIME-By spending more up-front design time creating a complete top-down design for the program, designers should be able to cut in half the time needed to test (and debug) the completed program. Less testing time means lower development coasts.

LESS MAINTENANCE TIME –All programs will need to be changed during their time of use. Programs that can be modified quickly and put back into use immediately will save the company time and therefore money.

These first two goals are both related to certain characteristics of structured designs. Structured designs are simple, clear, readable, and follow the same guidelines (within a company) no matter who did the design. Since all designers are following the same standards, any designer should be able to follow easily someone else's design. This concept, ego-less programming, means that a design should follow the style of the company, not an individual designer's style.

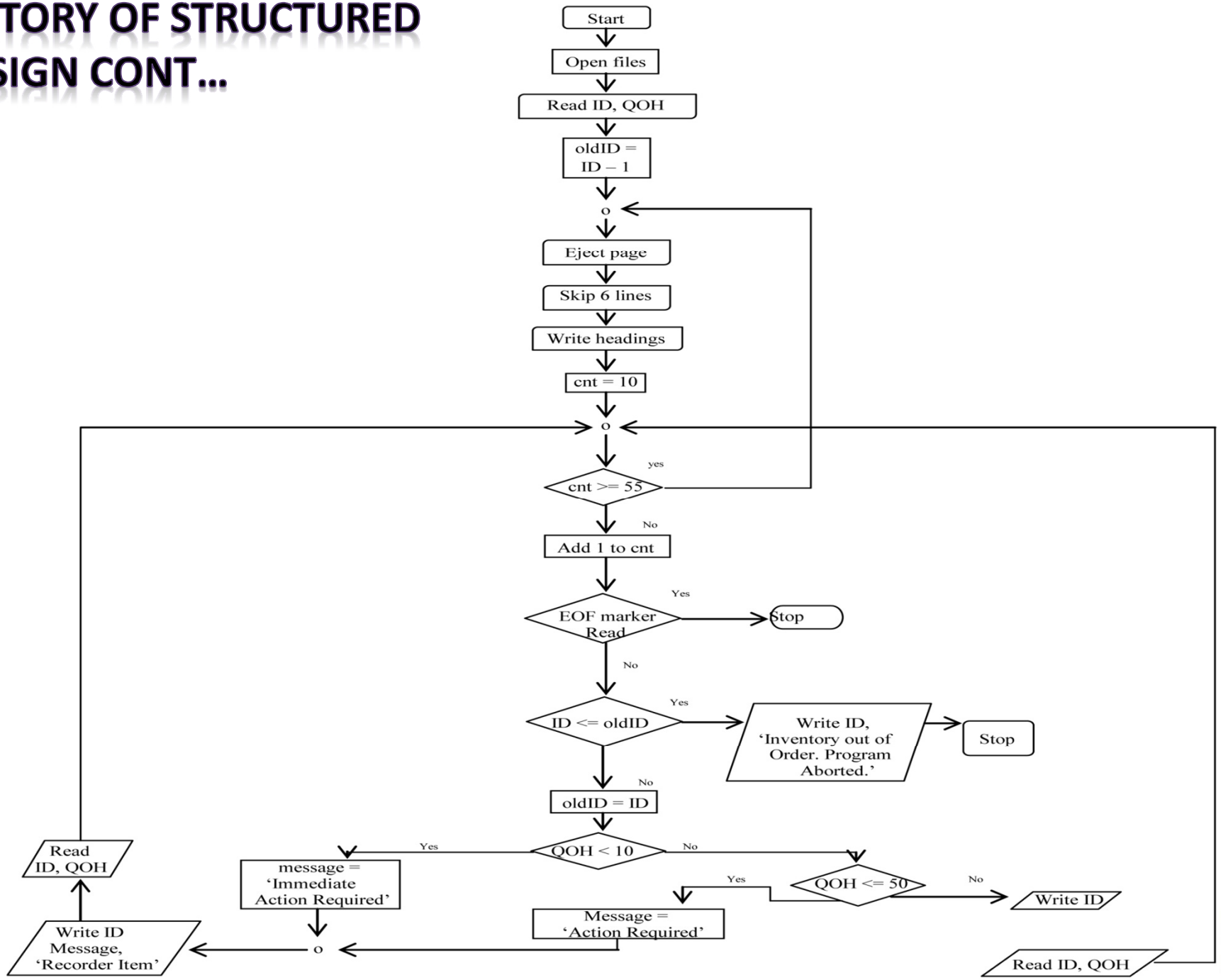
MAINTAINBILITY, NOT EFFICIENCY, AS PRIMARY GOAL-Program efficiency is no longer the final and major criterion for judging all programs as it once was. A program which executes very quickly but takes five times longer than other programs to modify every time a change is needed is not an efficient program. The first foal of design (and coding) is to make the design work through clear, simple, maintainable logic. Once that goal is met, the designer can investigate means to make the program run more efficiently as long as the efficiency does not come at the cost of the clarity and maintainability of the program. Because most business application programs follow our overall outline of something before the loop, a loop, and something after the loop, it is common for a very few lines to account for most of the execution time. Optimization of those few lines in the loop can have a major increase in efficiency without hurting the maintainability of the program.

HISTORY OF STRUCTURED DESIGN

A typical unstructured design of 25 years ago might look like the flowchart on the following page (although this is a very small design). This flowchart is design solution for a program to create an inventory list from a current holdings input file. The file is to be in order by inventory number and contains inventory number and quantity on hand for each item in inventory. The report should include all items in the input file unless an item is found to be out of order, in which case an error message should be written and the execution stopped.

If the quantity on hand is less than 10, the message “Immediate Action Needed” should be written along with a message to reorder the item. If the quantity on hand is 10 to 15, the message “Action Required” should be written along with a message to recorder the item. If the quantity on hand is greater than 50, the part number should be written with no message. “Reasonable” headings are also included in the report. The design is given in flowchart because we can show a GO TO using a flowline. Remember that pseudocode was developed along with structured design to correct some of the weakness of flowcharting. It would be much harder to create the same unstructured design in pseudocode.

HISTORY OF STRUCTURED DESIGN CONT…

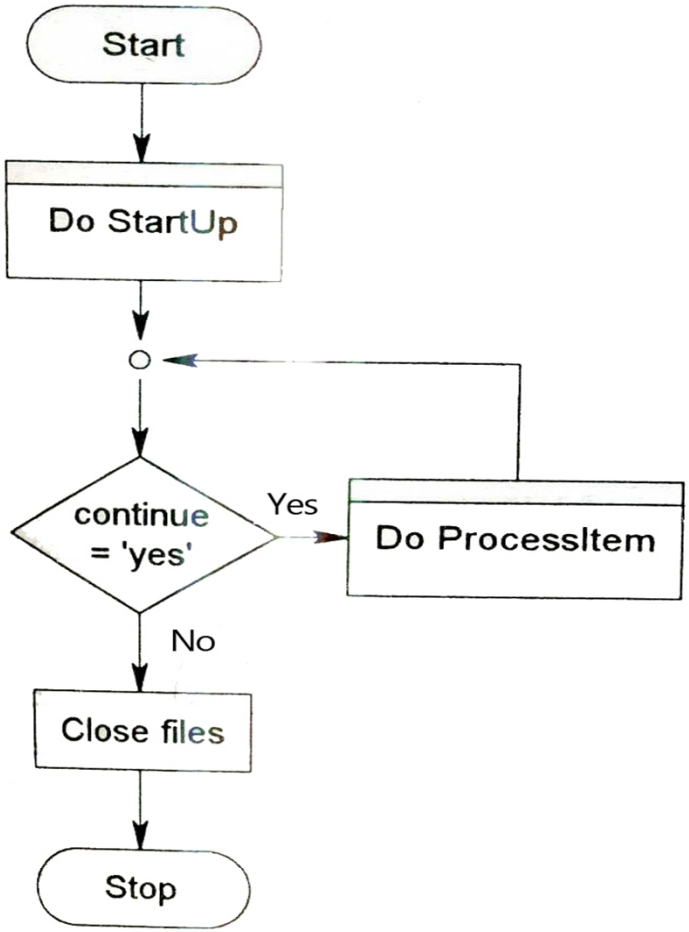


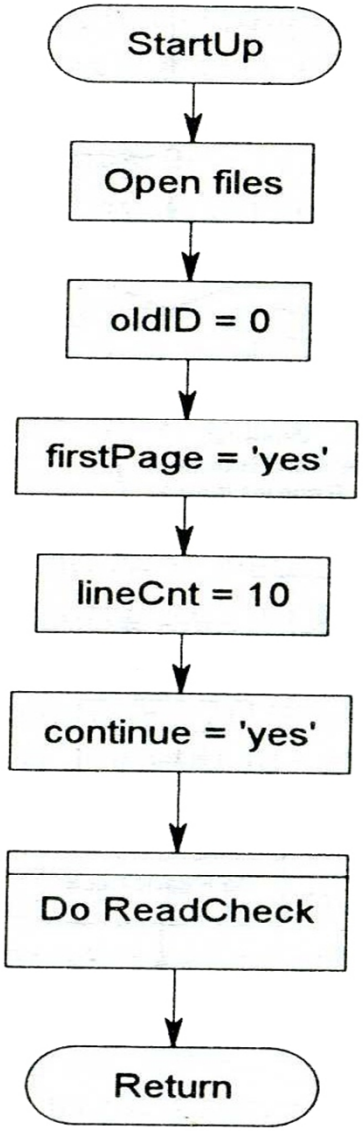
HISTORY OF STRUCTURED DESIGN CONT…

What's wrong. The design has two exits (two Stops). There are several selection structures and none has a really intended to act as a selection. The "real" loop test is not at the beginning or end of the loop, but in the middle. The design uses "GO TOs" to branch to other locations in the design without any return. In larger designs that included GO TO flowlines branching throughout the design, the page resembled a plate of spaghetti. That is why unstructured code was often referred to as "spaghetti code."

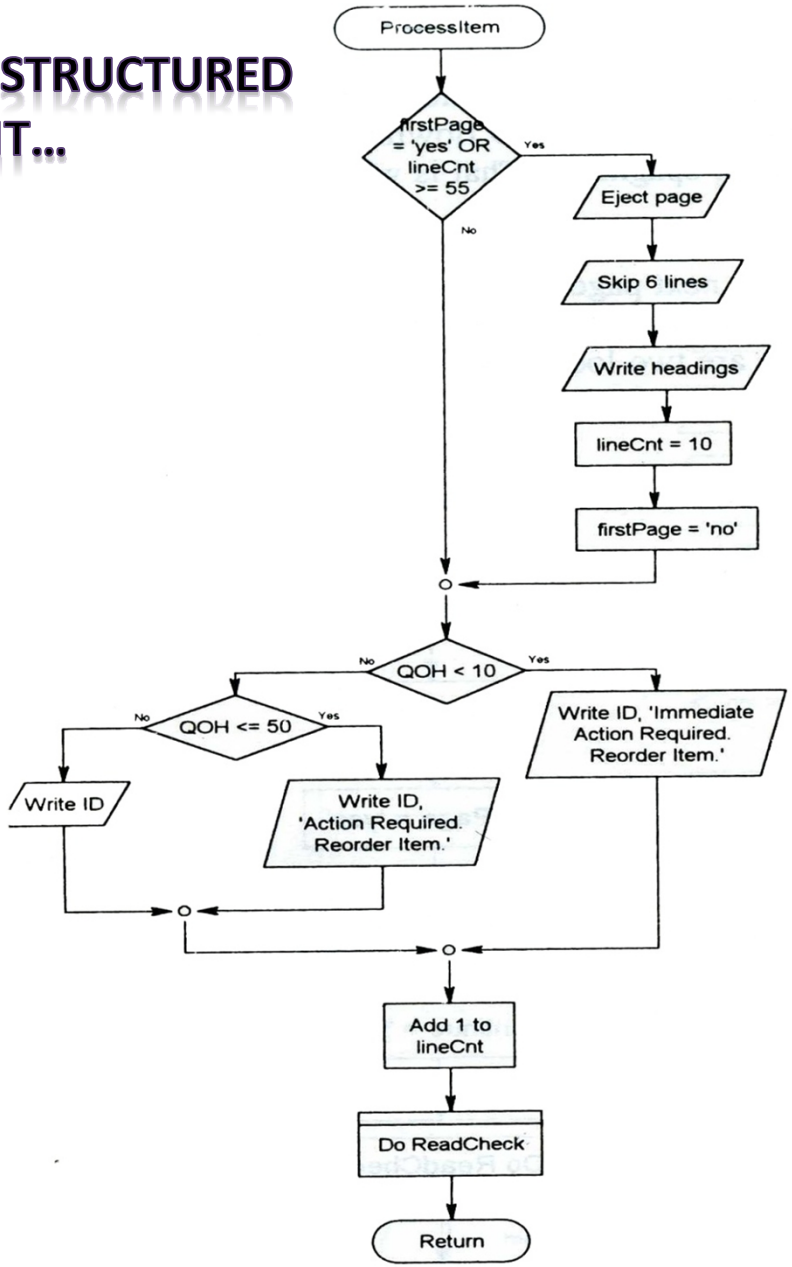
A structured solution follows below and on the next page. Notice that it uses modules to separate the details from the main structure. It uses a flag (continue) to control the loop. Since there are two reasons to exit the loop, there are two locations where the value of continue can be changed.

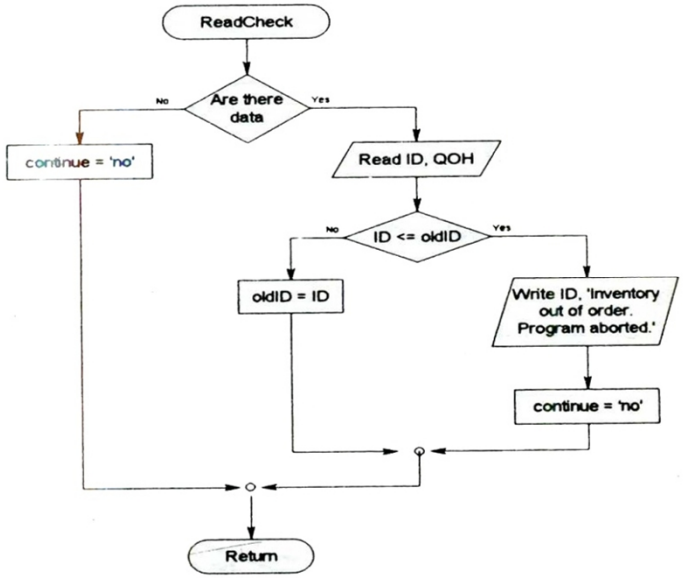
HISTORY OF STRUCTURED DESIGN CONT…





HISTORY OF STRUCTURED DESIGN CONT…





HISTORY OF STRUCTURED DESIGN CONT…

Try to take the image of the unstructured design and expand it into a much longer program. Programs that were thousands of lines long and still used the same unstructured “spaghetti code” were almost impossible to follow once they were written. In the early days of programming, programmers were left pretty much to themselves.

As long as their programs worked (and no matter how they worked), the programmers were left, rather like artist, to produce their creations. The problem was that their creations were so individualistic, so difficult to follow, and so poorly documented that maintaining them became an increasingly time-consuming task. Some programmers had to be retained on staff because they were the only ones who could modify the programmers they had written. In many companies, the backlog for maintenance became so great that there was no new program development. In most companies, the majority of programmer time was spent on program maintenance.

HISTORY OF STRUCTURED DESIGN CONT…

In 1964, Bohm and Jacopini, two mathematicians, presented a mathematical proof that all programming could be completed using the three basic structures (sequence, selection and loop). Conspicuously absent from their list of structures was the GO TO. Unfortunately, their paper did not receive much notice in the United States (for one thing, it was written in Italian). In 1966, an English translation was written in a U.S. journal, but the article still did not receive the attention it deserved. In 1968, E.W. Dijkstra wrote a letter to the editor of the *Association for Computing Machinery Journal* outlining the "harmful" effects of the use of the GO TO statement in programming. The letter served to focus attention on the ideas of structured design presented earlier by Bohm and Jacopini. From these beginnings, structured design was born, but it still didn't flourish.

In the early 1970s two major programming projects received major attention in the programming world. One was an IBM project completed for the *New York Times* and the other was a NASA project. Both utilized the concepts of structured design as we have outlined them here; both were major, multiyear project completed on time, within budget, and judged to be highly reliable. Structured design gained acceptance in the United Stated party because of the attention given to these two projects. By the mid-to-late seventies, structured design had become the standard design philosophy.

HISTORY OF STRUCTURED DESIGN CONT…

We are now going through a new upheaval in program design. Currently most programming is done with third-generation programming languages called *procedural languages*. With these languages we must define not only *what* is to be done, but also *how* it is to be done. The planning of the design is critical to defining the *how*. Newer, fourth-generation, nonprocedural languages take a different approach to program creation. With these languages it is no longer necessary to define *how*; the system establishes the how, once we tell it the *what*.

Does that mean program design is no longer going to be important? No! It means that program design will (as fourth generation languages become more prevalent) become more specialized for a few. But it is true now and will continue to be true that most student of design will never be programmers; they will utilize their skill in their work with application programs (spreadsheets, data base packages, etc.), in systems analysis, and other related occupations.

INTERACTIVE PROGRAMS

The programs we have designed so far are *batch* programs. The input is "collected" and stored in a file of some kind. When the program is executed, it opens the file and processes the input one record at a time. Once the command to execute the program is given, no further intervention form the user is needed because *all* the input are already gathered in the file. The program runs on its own, processing the stored batch of data.

In an *interactive* program, the user interacts with the computer during the execution of the program. Usually the interaction is in the form of providing the input. Data for a read statement would be typed in at the keyboard at the time it is needed for the program. There are many advantages (and some disadvantages) to interactive programs. For many purposes, interactive programs are telephoning in an order to a catalog store. When you provide the catalog number, the operator can check the availability of the item immediately using an interactive ordering program. If the item is not available, you can make a decision about ordering a different item, waiting until the item is available, or choosing a different color. Because the order program is interactive, you can make ordering decisions and modify the input while the order is being processed.

INTERACTIVE PROGRAMS CONT…

If the program were a batch program, your order would be stored in a file until the end of the day (or whenever the orders were processed), and all orders would be processed sequentially. A sequential process of the stored orders is actually a more efficient use of the computer, but a less satisfactory way for you to order. By the time the computer checks your order and finds that the item is out of stocks, it is too late for you to make a change. With interactive programs, we get timely input and output, and we get the opportunity to control the execution of the program based on our input.

For our perspective, interactive programs present some unique design considerations. When our input comes from the disk or some other computer-controlled peripheral, we don't have to worry much about the interface between the two pieces of hardware. Our program will give commands to the computer which will really them to the input/output devices. But when the input comes from a human being sitting at the keyboard, the interface is between the program/computer and the user. Now we have the program interfacing with the human, and it must be done so that the human understands what is needed and in what from. In a sense, the program is having a conversation with the human user. The program asks a question and the user answers it. The program must be designed so that it makes the best and clearest uses of the human user. There are two major design issues that concern us.

LOOP CONTROL

Most of our loops have been controlled indirectly by the end-of-file marker in the input file. We have used the loop test WHILE there are data which instructs the computer to check the data file to see if the input file contains another record or if the end-of-file marker is the next value to be read. When the input comes from the keyboard, there isn't an end-of-file marker because there really isn't a file. How will we control the loop? It needs to be controlled based on user input; essentially, does the user want to repeat the loop? One solution is to ask a question at the end of each loop ("Do you have more information to process?") and exit or repeat the loop based on the answer. This is a condition controlled loop much like a sentinel value. Perhaps we have an interactive college registration program that asks for a name and tells us to enter Quit as the name if we want to exit the program. The loop is controlled by the value of the name entered. If the name is "Quit," the program exits the One of the last statements within the loop would be to the next name. In both cases, the loop is controlled by user input. With the condition controlled loop that asks if the user wants to repeat the loop, a generalized task list might look like the following.:

LOOP CONTROL CONT…

REPEAT
    Prompt for input
    Read input
    Process input
    Process input
    Write output
    Ask user if loop should be repeated
    Read user response
UNTIL user says don't repeat loop

If the loop is controlled by a sentinel value, the task list will change because, as you know, sentinel value processing requires a priming read. If we use an UNTIL loop we are assuming that the first piece of input is not the sentinel value. All future input is then read at the bottom of the loop and is tested to see if the sentinel value has been entered. The generalized task list could be:

Prompt for input
Read input
REPEAT
    Process input
    Write input
    Prompt for input
    Read input
UNTIL user enters sentinel value

You probably noticed the use of UNTIL loops in the above task lists. With interactive input, UNTIL loops will be much more common. Remember that the UNTIL loop is always executed at least once since the controlling condition is not tested until *after* the first execution. Usually, we can assume the user wants to do the loop at least once in an interactive program. We will look in a moment at editing loops often seem more logical when designed as UNTIL loops. Any loop that can be done as an UNTIL loop, though, can also be done as a WHILE loop.

EDITING THE INPUT

When our input comes from a file, our design frequently just assumes that the input is in the correct form and is valid. But when the input comes from the keyboard, we must assume just the opposite. We must always account for the possibility that the user has made a typing error or skipped a line and typed in the wrong data, or mad some other input mistake. Most input will need to be "verified" by the user or checked against some standard before we accept it. The advantage is that because the program is interactive, invalid data can be corrected immediately. Much of this editing will be done is series of small UNTIL loops (read the name *until* you get a correct name; read the address *until* you get a correct address; read the course number *until* you get a valid course number, etc.).

EDITING THE INPUT CONT…

The more catastrophic an incorrect entry would be the more important it is to check the input. If a typo in a product number will only mean that a search is unsuccessful and the user gets a warning and a prompt to enter the next number, then the user can just correct the entry on the next loop iteration. But if the typo in the product number means that a customer is automatically charged for the purchase of some other product, some form of check needs to be done.

In the examples that follow, we will check virtually all the user input. Most users would consider such an extreme more of a hindrance that a help. As the designer you must be able to find an appropriate balance between checking the input to prevent errors and slowing down the user with constant prompts for verification. Where the balance point is will

The End of Section