CHAPTER 18.4
SOFTWARE

NATURE OF THE PROBLEM

Even though hardware costs have been declining dramatically over a 30-year period, the overall cost of developing and implementing new data processing systems and applications has not decreased. Because developing software is a predominantly labor-intensive effort, overall costs have been increasing. Furthermore, the problems being solved by software are becoming more and more complex. This creates a real challenge to achieve intellectual and management control over the software development process.

The successful development of software requires discipline and rigor coupled with appropriate management control arising from adequate visibility into the development process itself. This has led to the rise of *software engineering*, defined as the application of scientific knowledge to the design and construction of computer programs and the associated documentation and to the widespread use of standardized commercially available software packages. In addition, a set of software tools has been developed to assist in system analysis of designs. The tools, often called computer assisted systems engineering, or CASE tools, mechanize the graphic and textual descriptions of processes, test interrelationships, and maintain cross-referenced data dictionaries.

THE SOFTWARE LIFE-CYCLE PROCESS

In the earlier history of software the primary focus was on its development, but it has become evident that many programs are not one-shot consumables but are tools intended to be used repetitively over an extended time. As a result, it is obvious that the entire software life cycle must be considered. The software life cycle is that period of time over which the software is defined, developed, and used. Figure 18.4.1 shows the traditional model of the software life-cycle process and its five major phases. It begins with the *definition phase*, which is the key to everything that follows. During the definition phase, the system requirements to be satisfied by the system are developed and the system specifications, both hardware and software, are developed. These specifications describe *what* the software product must accomplish. At the same time, test requirements should also be developed as a requisite for systems acceptance testing.

The *design phase* is concerned with the design of a software structure that can meet the requirements. The design describes *how* the software product is to function. During the *development phase*, the software product is itself produced, implemented in a programming language, tested to a limited degree, and integrated. During the *test phase*, the product is extensively tested to show that it does in fact satisfy the user's requirements. The *operational phase* includes the shipment and installation of the data-processing system in the user's facility. The system is then employed by the user, who usually embarks on a maintenance effort, modifying the system to improve its performance and to satisfy new requirements. This effort continues for the remainder of the life of the system.

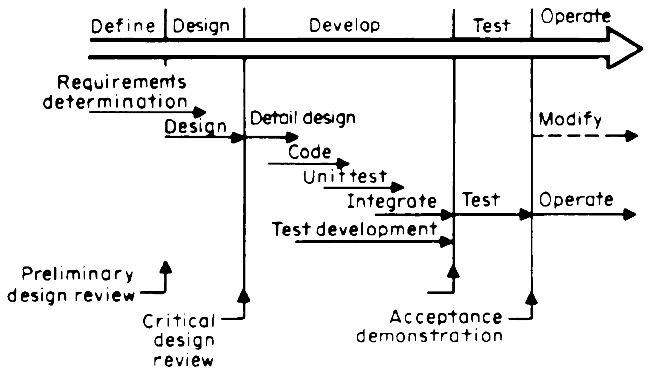


FIGURE 18.4.1 Traditional model of the software life-cycle process showing its five major phases.

PROGRAMMING

When a stored-program digital computer operates, its storage contains two types of information: the data being processed and program instructions controlling its operations. Both types of information are stored in binary form. The control unit accesses storage to acquire instructions; the ALU makes reference to storage to gain access to data and modify it. The set of instructions describing the various operations the computer is designed to execute is referred to as a *machine language*, and the act of constructing programs using the appropriate sequences of these computer instructions is called *machine-language programming.* It is possible but expensive to write them directly in machine languages, and maintenance and modification is virtually impossible. *Programming languages* have been created to make the code more accessible to its writers.

A programming language consists of two major parts: the language itself and a translator. The language is described by a set of *symbols* (the *alphabet*) and a *grammar* that tells how to assemble the symbols into correct strings. The *translator* is a machine-language program whose main function is to translate a program written in the *programming language* (the *source code*) into machine language (*object code*) that can be executed in the computer. Before describing some of the major programming languages currently in use, we consider two important programming concepts, *alternation and iteration*, and also see by examples some of the difficulties associated with machine-language programming.

ALTERNATION AND ITERATION

These techniques are illustrated here using a computer whose storage consists of 10,000 words each containing 4 bytes numbered 1 to 4. The instruction format is

1	2	3 4
op code	0	Address

Op code	Name	Description
01	LOAD	Loads value from addressed word into data register
02	COMP	Compares value of addressed word with data-register value
03	ADD	Adds value of addressed word to data register
08	STORE	Copies value of data register into addressed storage word
20	BRLO	Branches if data-register value from last previously executed COMP was less than comparand

The computer used in this simplified example contains a separate nonaddressable data register that contains one word of data. Further, each instruction is accessed at an address 1 more than that of the previously executed instruction unless that instruction was BRLO instruction with a low COMP condition, in which case the address part of the BRLO instruction is the address at which the next instruction is to be accessed.

Consider the following program instructions (beginning at address 0100) to select the lower value of two items (in words 0950 and 0951) and place the selected value in a specific place (word 0800):

Address	Instruction	Effect
0100	01000950	Place first-item value in data register
0101	02000951	Compare second-item value with data-register value
0102	20000104	Branch to next instruction at address 0104 if data-register value was lower
0103	01000951	Place second item value in data register
0104	08000800	Store lower value in result (word 0800)

FLOWCHARTS

One way to depict the logical structure of a program graphically is by the use of *flowcharts*. Flowcharts are limited in what they can convey about a computer program, and with the advent of modern programming design languages they are becoming less widely used. However, they are used here to portray these simple programs graphically. The program of the preceding example is depicted by the flowchart shown in Fig. 18.4.2.

The flowchart contains boxes representing processes (rectangular boxes) and decisions (alternations—diamond-shaped boxes). The arrows connecting the boxes represent the paths and sequences of instruction execution. An alternation represents an instruction (or a sequence of instructions) with more than one possible successor depending on the result of some processing test (this is commonly a conditional branch). In the example, instruction 0103 is or is not executed depending on the values of the two items.

If the example is extended to require finding the least of four item values, the flowchart is that shown in Fig. 18.4.3. If the example is further extended to find the largest value of 1000 items (in locations 0336 through 0790 inclusive in hexadecimal), the flowchart and the corresponding program become very large if analogous extensions of the flowcharts are used.

The alternative is to use the technique known as the *program loop*. A program loop for this latter example is:

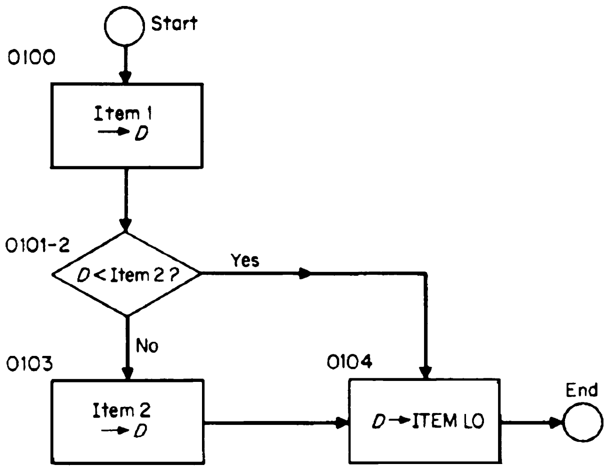


FIGURE 18.4.2 Flowchart of a simple program. The boxes represent processes; the diamonds represent decisions.

Address	Instruction	Effect
0100	01000950	Move first item as initial value of result (ITEMHI)
0101	08000800	
0102	01000900	Initialize loop to begin with item 2
0103	08000104	
0104	(00000000)	Loop, *N*th item to data register
0105	02000800	Compare with prior ITEMHI value
0106	20000108	Branch to 108 if *N*th item value low
0107	08000800	Store *N*th item value as ITEMHI
0108	01000104	Increment value, of *N* by 1
0109	03000901	
010A	08000104	
010B	02000902	Compare against $N = 1001$
010C	2000104	Branch for looping if $N < 1001$
010D	end	
0900	01000951	Load item 2; initial instruction
0901	00000001	Address increment of 1
0902	010003E9	Limit test; load 1001st item

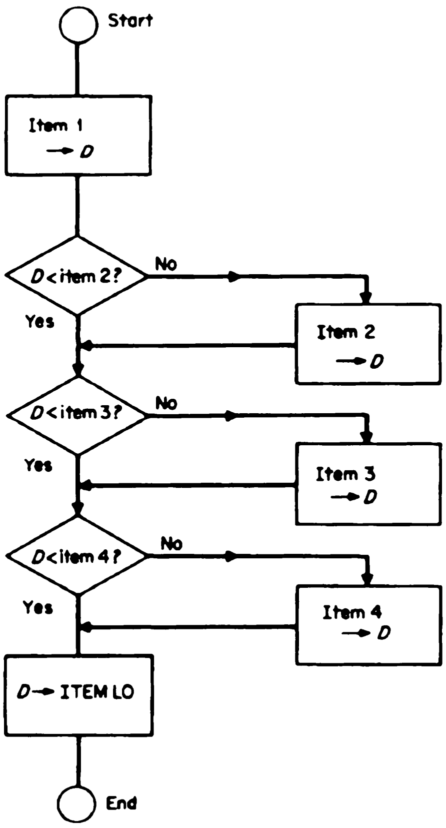


FIGURE 18.4.3 Flowchart of a repetitive task.

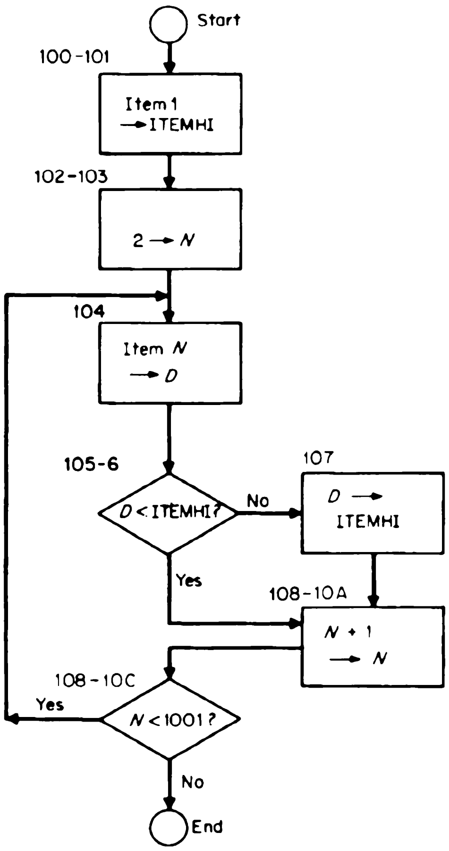


FIGURE 18.4.4 Flowchart showing a program loop.

The corresponding flowchart appears in Fig. 18.4.4. The loop proper (instructions 0104 to 010C) is executed 999 times. The instruction at 0104 accesses the *N*th item and is indexed each time the program flows through the loop so that on successive executions successive words of the item table are obtained. After each loop execution, a test is made to determine whether processing is complete or a branch should be made back to the beginning of the loop to repeat the loop program.

The loop proper is preceded by several instructions that *initialize* the loop, presetting ITEMHI and the instruction 0104 value for the first time through. A loop customarily has a *process* part, and an *induction* part to make changes for the next loop iteration, and an *exit test* or termination to determine whether an additional iteration is required.

ASSEMBLY LANGUAGES

The previous example illustrates the difficulty of preparing and understanding even simple machine-language programs. One help would be the ability to use a symbolic (or mnemonic) representation of the operations and addresses used in the program. The actual translation of these symbols to specific computer operations and addresses is a more or less routine clerical procedure. Since computers are well suited to performing such routine operations, it was quite natural that the first automatic programming aids, *assembly languages* and their associated assembly programs, were developed to take advantage of that fact. Assembly languages permit the critical addressing interrelations in a program to be described regardless of the storage arrangement, and they can produce therefrom a set of machine instructions suitable for the specific storage layout of the computer in use. An assembly-language program for the 1000-value program of Fig. 18.4.4 is shown in Fig. 18.4.5.

The program format illustrated is typical. Each line has four parts: location, operation, operand(s), and comments. The location part permits the programmer to specify a symbolic name to be associated with the address

LOC	OP	Operand	Comment
	ORG	100	Start at ADDR 100
START	LOAD	ITEM 1	Move first item value to ITEMHI
	STORE	ITEM HI	
	LOAD	CONST 1	Set LOOP to start with second item
	STORE	LOOP ST	
LOOP ST	CONST		• Load *N*TH item
	COMP	ITEM HI	Compare with previous HI
	BRLO	LOOP INC	Skip if lower
	STORE	ITEM HI	Store on equal or high
LOOP INC	LOAD	LOOP ST	Increment *N* by 1
	ADD	ONE	
	STORE	LOOP ST	Modify storage
	COMP	CONST 2	Exit test
	BRLO	LOOP ST	Repeat if $N < 1001$
	. . .		(End of example)
	ORG	700	
CONST 1	LOAD	ITEM 1 + 1	Initial LOOP ST instruction
ONE	CONST	+1	Incrementation constant
CONST 2	LOAD	ITEM 1 + 1000	LOOP end test constant
	. . .		
	ORG	800	
ITEM HI	RESRV	1	Word where high value left
	. . .		
	ORG	950	
ITEM 1	RESRV	1	First item value
	RESRV	999	Second through thousandth items

FIGURE 18.4.5 An assembly program. The program statements are in a one-to-one correspondence with machine instructions. Hence the procedure is fully supplied by the programmer according to the particular macroinstruction set of the system. The assembly language alleviates housekeeping routines, such as specific assignments, and makes user-oriented symbols possible instead of numeric or binary code.

of the instruction (or datum) defined on that line. The operation part contains a mnemonic designation of the instruction operation code. Alternatively, that line may be designated to be a datum constant, a reservation of data space, or a designation of an assembly *pseudo operation* (a specification to control the assembly process itself). Pseudo operations in the example are ORG for origin and END to designate the end of the program.

The operand field(s) give the additional information needed to specify the machine instruction, e.g., the name of a constant, the size of the data reservation, or a name associated with a pseudo operation. The comment part serves for documentation only; it does not affect the assembly-program operation.

After a program is written in assembly language, it is processed by an assembler. The assembly program reads the symbolic assembly-language input and produces (1) a machine instruction program with constants, usually in a form convenient for subsequent program loading, and (2) an assembly listing that shows in typed or printed form each line of the symbolic assembly-language input, together with any associated machine instructions or constants produced therefrom.

The assembly pseudo operation ORG specifies that the instructions and/or constant entries for succeeding lines are to be prepared for loading at successive addresses, beginning at the specified load origin (value of operand field or ORG entry). Thus the 13 symbolic instructions following the initial ORG line in Fig. 18.4.5 are prepared for loading at addresses 0100 through 010C inclusive, with the following symbolic associations established:

Location symbol	(Local) address
START	100
LOOP ST	104
LOOP INC	108

Four instructions of this group of 13 contain the symbol LOOP ST in the operand field, and the corresponding machine instructions will contain 0104 in their address parts.

The operation of a typical assembly program therefore consists of (1) collecting all location symbols and determining their values (addresses), called *building the symbol table*, and (2) building the machine instructions and/or constants by substituting op codes for the OP mnemonics and location symbol values for their positions in the operand field. The symbol table must be formed first since, as the first instruction in the example shows, a machine instruction may refer to a location symbol that appears in the location field near the program end. Thus most assembly programs process the program twice; the *first pass* builds the symbol table, and the *second pass* builds the machine-language program. Note in the example the use of the operation RESRV to reserve space (skipping in the load-address sequence) for variable data.

Assembly language is specific to a particular computer instruction repertoire. Hence, the basic unit of assembly language describes a single machine instruction (so-called one-for-one assembly process).

Most assembly languages have a *macroinstruction* facility. This permits the programmer to define *macros* that can generate desired sequences of assembly-language statements to perform specific functions. These macro definitions can be placed in macro libraries, where they are available to all programmers in the facility.

The term *procedure* (also *subroutine* and *subprogram*) is used to refer to a group of instructions that perform some particular function used repeatedly in essentially the same *context.* The quantities that vary between contexts may be regarded as parameters (or arguments) of the procedure. The method of adaptation of the procedure determines whether it is an *open* or *closed* procedure.

An open subroutine is adapted to its parameter values during code preparation (assembly or compilation) in advance of execution, and a separate copy of the subroutine code is made for each different execution context. A closed subroutine is written to adapt itself during execution to its parameter values; hence, a single copy suffices for several execution contexts in the same program. The open subroutine executes faster since tailoring to its parameter values occurs before execution begins. The closed subroutine not only saves storage space, since one copy serves multiple uses, but is more flexible, in that parameter values derived from the execution itself can be used.

A closed subroutine must be written to determine its parameter values in a standard way (including the return point after completion). The conventions for finding the values and/or addresses of values are called the subroutine linkage *conventions.* Quite commonly, a single address is placed in a particular register, and this address in turn points to a consecutively addressed list of addresses and/or values to be used. Subroutines commonly use (or *call*) other closed subroutines, so that there are usually a number of levels of subroutine control available at any point during execution. That is, one routine is currently executing, and others are waiting at various points in partially executed condition.

HIGH-LEVEL PROGRAMMING LANGUAGES

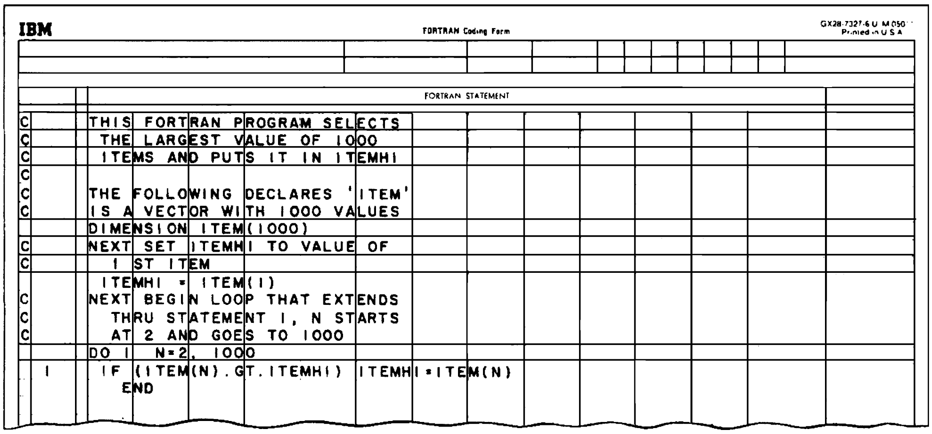
On general-purpose digital computers, *high-level programming languages* have largely superseded assembly languages as the predominant method of describing application programs. Such programming languages are said to be *high-level* and *machine-independent.* High-level means that each program function is such that several or many machine instructions must be executed to perform that function. Machine-independent means that the functions are intended to be applied to a wide range of machine-instruction repertoires and to produce for each a specific machine representation of data.

The high-level language translator is known as a *compiler*, i.e., a program that converts an input program written in a particular high-level language (*source program*) to the machine language of a particular machine type (*object program*) each time the source code is executed.

HIGH-LEVEL PROCEDURAL LANGUAGES

Most of the high-level programming languages are said to be *procedural.* The programmer writing in a high-level procedural language thinks in terms of the precise sequence of operations, and the program description is in terms of sequentially executed *procedural statements.* Most high-level procedural languages have statements for *documentation, procedural execution, data declaration*, and various compiler and execution *control specifications.*

The program in Fig. 18.4.6, written in the FORTRAN high-level language, describes the program function given in Fig. 18.4.5 in assembly language. The first six lines are for documentation only, as indicated by C in the first column. The DIMENSION statement defines ITEM to consist of 1000 values. The assignment statement ITEMHI = ITEM (1) is read as "set the value of ITEMHI to the value of the first ITEM." The next statement is a loop-control statement meaning: "do the following statements through the statement labeled 1 for the variable *N* assuming every value from 2 through 1000." The statement labeled 1 causes a test to be made to "see if the *N*th ITEM is greater than .GT. the value of ITEMHI, and if so, set the ITEMHI value equal to the value of the *N*th item.


IBM FORTRAN Coding Form GX28-7327-6 U M 050 Printed in U.S.A.

FORTRAN STATEMENT

```
C       THIS FORTRAN PROGRAM SELECTS
C        THE LARGEST VALUE OF 1000
C        ITEMS AND PUTS IT IN ITEMHI
C
C       THE FOLLOWING DECLARES 'ITEM'
C       IS A VECTOR WITH 1000 VALUES
        DIMENSION ITEM(1000)
C       NEXT SET ITEMHI TO VALUE OF
C         1 ST ITEM
         ITEMHI = ITEM(1)
C       NEXT BEGIN LOOP THAT EXTENDS
C         THRU STATEMENT 1, N STARTS
C         AT 2 AND GOES TO 1000
        DO 1  N=2, 1000
    1    IF (ITEM(N).GT.ITEMHI) ITEMHI=ITEM(N)
          END
```

FIGURE 18.4.6 An example of a FORTRAN program, corresponding to the flowchart of Fig. 18.4.4 and assembly program of Fig. 18.4.5.

FORTRAN

The high-level programming languages most commonly used in engineering and scientific computation are C++, FORTRAN, ALGOL, BASIC, APL, PL/I, and PASCAL FORTRAN, the first to appear was developed during 1954 to 1957 by a group headed by Backus of IBM. Based on algebraic notation, it allows two types of numbers: integers (positive and negative) and floating point. Variables are given character names of up to six positions. All variables beginning with the letters I, J, K, L, M, or N are integers; otherwise they are floating point. Integer constants are written in normal fashion, 1, 0, –4, and so on. Floating-point constants must contain a decimal point, 3.1, –0.1, 2.0, 0.0, and so on. For example, 6.02×10^{24} is written 6.02E24. This standard notation was adopted to accommodate the limited capability of computer input-output equipment.

READ and WRITE statements permit values of variables to be read into or written from the ALU, from or to input, output, or intermediate storage devices. The latter may operate merely by transcribing values or may be accompanied by conversions or editing specified in a separate FORMAT statement. Some idea of the range of operations provided in FORTRAN is shown by the following value-assignment statement:

```
ROOT = (–(B/2.0) + SQRT ((B/2.0) ** 2 – A*C))/A
```

This is the formula for the root of a quadratic equation with coefficients A, B, and C. The asterisk indicates multiplication, / stands for division, and ** exponentiation.

The notation: *name* (*expression*) and *name* (*Expression, expression*), and so forth, is used in FORTRAN with two distinct meanings depending on whether or not the specific name appears in a DIMENSION statement. If so, the *expression*(s) are subscript values; otherwise the *name* is considered to be a function name, and the expressions are the values of the arguments of the function. SQRT ((B/2.0) **2 – A*C) in the preceding assignment statement requires the expression (B/2.0)**2 – A*C to be evaluated, and then the function (square root here) of that value is determined. Square root and various other common trigonometric and logarithmic functions and their respective inverses are standardized in FORTRAN, typically as closed subroutines.

The same notations may be employed for a function defined by a FORTRAN programmer in the FORTRAN language. This operation is performed by writing a separate FORTRAN program headed by the statement

```
FUNCTION name (arg 1, arg 2, etc.)
```

where arg represents the name that stands for the actual argument value at each evaluation of that function. Similarly, any action or set of actions described by a closed FORTRAN subroutine is called for by “CALL subroutines (args)” together with a defining FORTRAN subroutine headed by “SUBROUTINE subroutine name (args).”

BASIC

BASIC is high-level programming language based on algebraic notation that was developed for solving problems at a terminal; it is particularly suitable for short programs and instructional purposes. The user normally remains at the terminal after entering his program in BASIC, while it compiles, executes, and types the output, a process that typically requires only a few seconds. Widely used in PCs, BASIC is usually bundled with PC operating systems. It is available in both interpretive and compiled versions, and may include an extensive set of programming capabilities. Visual BASIC is currently being used on display-oriented operating systems such as Windows.

APL

*A p*rogramming *l*anguage (APL) is high-level language that is often used because it is easy to learn and has an excellent interactive programming system supporting it. Its primitive objects are arrays (lists, tables, and so forth). It has a simple syntax, and its semantic rules are few. The usefulness of the primitive functions is further enhanced by operations that modify their behavior in a systematic manner. The sequence control is simple because one statement type embraces all types of branches and the termination of the execution of any function always returns control to the point of use. External communication is established by variables shared between APL and other systems.

PASCAL

An early high-level programming languages is PASCAL, developed by Niklaus Wirth. It has had widespread acceptance and use since its introduction in the early 1970s. The language was developed for two specific purposes: (1) to make available a language to teach programming as a systematic discipline and (2) to develop a language that supports reliable and efficient implementations. PASCAL provides a rich set of both control statements and data structuring facilities. Six control statements are provided: BEGIN-END, IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL, FOR-DO, and CASE-END. Similar control statements can be found in virtually all high-level languages.

In addition to the standard scalar data types, PASCAL provides the ability to extend the language via user-defined scalar data types. In the area of higher-level structured data types, PASCAL extends the array facility of ALGOL 60 to include the record, set, file, and pointer data types. In addition to these, PASCAL contains a number of other features that make it useful for programming and teaching purposes. In spite of this, PASCAL is a systematic language and modest in size, attributes that account for its popularity.

ADA PROGRAMMING LANGUAGE

(Ada is a registered trademark of the Department of Defense.) Ada is named after Lord Byron's daughter. This language was developed by the U.S. Department of Defense to be a single successor to a number of high-level languages in use by the armed forces of the United States. It was finalized in 1980.

The Ada language was designed to be a strongly typed language, with features from modern programming language theory and software engineering practices. It is a block-structured language providing mechanisms for data abstraction and modularization. It supports concurrent processing and provides user control over scheduling and interrupt handling.

C PROGRAMMING LANGUAGE

Research is continuing in the development of new languages that support the concepts growing out of modern software technology development. One such language is the C programming language (a registered trademark of AT&T). C is a general-purpose programming language designed to feature modern control flow and data structures and a rich set of operators, yet provide an economy of expression. Although it was not specialized for any one area of application, it has been found especially useful for implementing operating systems and is being more widely used in communications and other areas. C++ is a later version of this language. An extension of this language has been called JAVA, developed by Sun Microsystems.

OBJECT-ORIENTED PROGRAMMING LANGUAGES

A second area of programming language development has been the creation of object-oriented languages. These are used for message-object programming, which incorporates the concepts of objects that communicate by messages. An object includes data, a set of procedures (methods) that operate on the data, and a mechanism for translating messages. These languages should contribute to improved reusability of software—a highly sought after goal to increase programming productivity and reduce software costs. These languages are based on the concept that "objects," once defined, are henceforth available for reuse, without reprogramming. Programs can then be viewed as mechanisms that employ the appropriate object at the appropriate time to accomplish the task at hand.

COBOL AND RPG

High-level programming languages used for business data-processing applications emphasize description and handling of files for business record keeping. Two widely used programming languages for business applications are COBOL (*co*mmon *b*usiness-*o*riented *l*anguage) and RPG (*r*eport *p*rogram *g*eneration). Compilers for these languages, with generalized sorting programs, form the fundamental automatic programming aids of many computer installations primarily used for business data processing. COBOL and RPG have comparable file, record, and field-within-record descriptive capabilities, but the specification of processing and sequence control device from basically different concepts.

OPERATING SYSTEMS

There are many reasons for developing and using an *operating system* for a digital computer. One of the main reasons is to optimize the scheduling and use of computer resources, so as to increase the number of jobs that can be run in a given period. Creation of a multiprogramming environment means that the resources and facilities of the computing system can be shared by a number of different programs, each written as if it were the only program in the system.

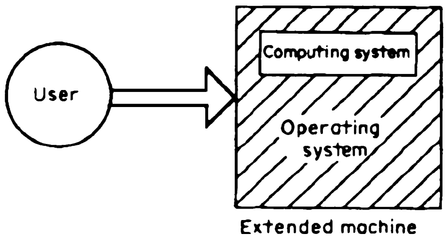


FIGURE 18.4.7 The user's view of the operating system as an extension of the computing system yet an integral part of it.

Another major objective for an operating system is to provide the full capability of the computing system to the user while minimizing the complexity and depth of knowledge of the computer system required. This is accomplished by establishing standard techniques for handling system functions like *program calling* and *data management* and providing a convenient and effective interface to the user. In effect the user is able to deal with the operating system as an entity rather than having to deal with each of the computer's features. As indicated in Fig. 18.4.7 each user will be thought of conceptually as a unit consisting of both the hardware and the programs and procedures that make up the operating system.

GENERAL ORGANIZATION OF AN OPERATING SYSTEM

There are many ways to structure operating systems, but for the purpose of this discussion the organization shown in Fig. 18.4.8 is typical. The operating system is composed of two major sets of programs, control (or supervision) programs and processing programs. *Control programs* supervise the execution of the support programs (including the user application programs), control the location, storage, and retrieval of data, handle interrupts, and schedule jobs and resources needed in processing. *Processing programs* consist of language translators, service programs, and user-written application programs, all of which are used by the programmer in support of program development.

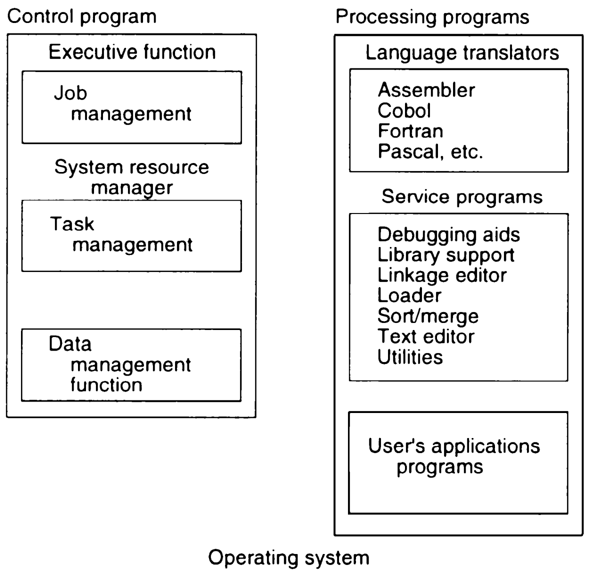


FIGURE 18.4.8 A typical operating system and its constituent parts.

The work to be processed by the computer can be viewed as a stack of jobs to be run under the management of the control program. A *job* is a unit of computational work that is independent of all other jobs concurrently in the system. A single job may consist of one or a number of *steps.*

TYPES OF OPERATING SYSTEMS

There are many basic types of operating systems including multiprogramming, time-sharing, real-time, and multiprocessing.

The multiprocessing system must schedule and control the execution of jobs that are distributed across two or more coupled processors. These processors may share a common storage, in which case they are said to be *tightly* (or *directly*) *coupled*, or they may have their own private storage and communicate via other means such as sending messages over networks, in which case they are said to be *loosely coupled.*

Operating systems were generally developed for a specific CPU architecture or for a family of CPUs. For example, MS-DOS and WINDOWS apply to xx86-based systems. However, one operating system, the UNIX system (a registered trademark of AT&T), has been *transported* to a number of different manufactures' systems and is in very wide use today. UNIX was developed as a unified, interactive, multiuser system. It consists of a kernel that schedules tasks and manages data, a shell that executes user commands—one at a time or in a series called a pipe—and a series of utility programs.

TASK-MANAGEMENT FUNCTION

This function, sometimes called the *supervisor*, controls the operation of the system as it executes units of work known as *tasks* or *processes.* (The performance of a task is requested by a job step.) The distinction between a task and a program should be noted. A *program* is a static entity, a sequence of instructions, while a *task* is a dynamic entity, the work to be done in execution of the program. Task management initiates and controls the execution of tasks. If necessary it controls their synchronization. In allocates system resources to tasks and monitors their use. In particular it is concerned with the dynamic allocation and control of main storage space.

Task management handles all interrupts occurring in the computer, which can arise from five different sources: (1) supervisor call interrupts occur when a task needs a service from the task-management function, such as initiating an I/O operation; (2) program interrupts occur when unusual conditions are encountered in the execution of a task; (3) I/O interrupts indicate that an I/O operation is complete or some unusual condition has occurred; (4) machine-check interrupts are initiated by the detection of hardware errors; and (5) external interrupts are initiated by the timer, by the operator's console, or other external devices.

DATA MANAGEMENT

This function provides the necessary I/O control system services needed by the operating system and the user application programs. It frees the programmer from the tedious and error-prone details of I/O programming and permits standardization of these services. It constructs and maintains various file organization structures, including the construction and use of index tables. It allocates space on disc (auxiliary) storage. It maintains a directory showing the locations of data sets (files) within the system. It also provides protection for data sets against unauthorized entry.

OPERATING SYSTEM SECURITY

One of the major concerns in the design of operating systems is to make certain that they are reliable and that they provide for the protection and the integrity of the data and programs stored within the system. Work is under way to develop secure operating systems. These systems use the concept of a security kernel—a minimal set of operating system programs that are formally specified and designed so that they can be proved to implement the desired security policy correctly. This assures the corrections of all access-controlling operations throughout the system.

SOFTWARE-DEVELOPMENT SUPPORT

There have been great strides in software engineering technology. Out of the research and development efforts in universities, industry, and government have emerged a number of significant ideas and concepts that can have significant and long-lasting influence on the way that software is developed and managed. Those concepts are just now starting to find their way into the software-development process but should become more widely used in the future. They are briefly reviewed below.

REQUIREMENTS AND SPECIFICATIONS

This has been one of the problem areas through the years. Analysis and design errors are, by far, the most costly and crucial types of errors, and a number of attempts are being made to develop methods for recording and analyzing software requirements and developing specifications. Most requirements and specifications documents are still recorded in English narrative form, which introduces problems of inconsistency, ambiguity, and incompleteness. These problems are addressed with CASE tools and structured programming.

SOFTWARE DESIGN

The work of Dijkstra, Hoare, and Mills (1968, 1976) had a major influence on software-design methodology by introducing a number of concepts that led to the development of *structured programming*. Structured programming is a methodology based on mathematical rigor. It uses the concept of top-down design and

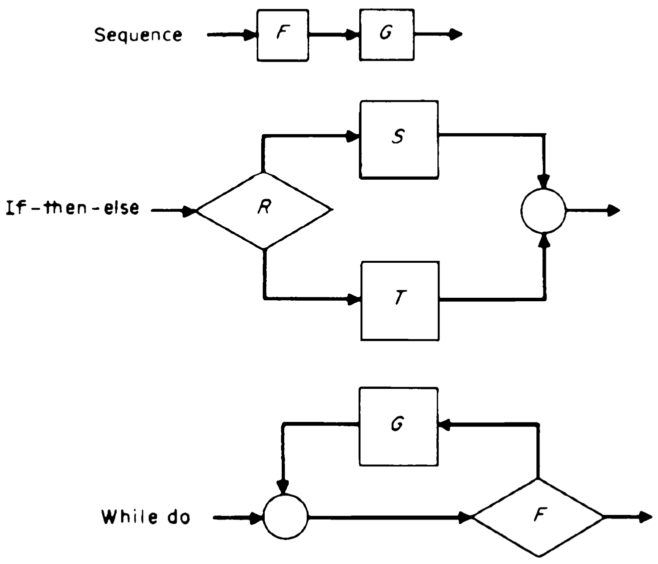


FIGURE 18.4.9 Basic set of structured control primitives.

implementation by describing the program at a high level of abstraction and then expanding (refining) this abstraction into more detailed representations through a series of steps until sufficient detail is present for implementation of the design in a programming language to be possible. This process is called *stepwise refinement.* The design is represented by a small, finite set of primitives, such as shown in Fig. 18.4.9. These three primitives are adequate for design purposes but for convenience several others have been introduced; namely, the *indexed alternation* or *case*, the *do-until*, and the *indexed sequence* or *for-do* structure.

It also recognized that the organization and representation of software systems are clearer if certain data and the operations permitted on those data are organized into data abstractions. The internal details of the organization of the data are hidden from the user.

The result of applying this methodology is the organization of a sequential software process into a hierarchical structure through the use of stepwise refinement. The software system structure is then defined at three levels—the *system*, the *module*, and the *procedure.* The system (or *job*) describes the highest level of program execution. The system is decomposed into modules. The module is composed of one or more procedures and data that persist between successive invocations of the module. The procedure is the lowest level of system decomposition, the executable unit of the stored program.

Another important aspect of the design process is its documentation. There is a critical need to record the design as it is being developed, from its highest level all the way to its lowest level of detail, before its implementation in a programming language, using language that can be used not only to communicate software designs between specialists in software development but also between specialists and nonspecialists in rigorous logical terms.

Important elements of the software development process are reviews, walk-throughs, and inspections, which can be applied to the products of the software-development process to ensure that they are complete, accurate, and consistent. They are applied to such areas as the design (design inspections), the software source code (code inspections), documentation, test designs, and test-results analysis. The goals of the software review and inspection process are to ensure that standards are met, to check on the quality of the product, and to detect and correct errors at the earliest possible point in the software life cycle. Another important value of the review process is that it permits progress against development-plan milestones to be measured more objectively and rework for error correction to be monitored more closely.

TESTING

An important activity in the software-development cycle that is often ignored until too late in the process is testing. It is also important to note what testing can and cannot do for the software product. Quality cannot be tested into the software; it must be designed into it. Test planning begins with the requirements analysis at the beginning of the project. Requirements should be testable; i.e., they should be stated in a form that permits the final product to be tested to assure that it satisfies the requirements. Test planning and test designs should be developed in parallel with the design of the software.

EXPERT SYSTEMS

One important area of research in computer science has been that of *artificial intelligence.* The most successful application of artificial intelligence techniques has been in the development of *expert systems*, or *knowledge-based systems*, as they are often called. These are human-machine interactive systems with specialized problem-solving expertise that are used to solve complex problems in such specific areas as medicine, chemistry, mathematics, and engineering. This expertise consists of knowledge about the particular problem domain that the expert system is designed to support (e.g., diagnosis and therapy for infectious diseases) and planning and problem-solving rules for processes used to identify and solve the particular problem at hand. The two main elements of an expert system are its knowledge base, which contains the domain knowledge for the problem area being addressed, and the inference engine, which contains the general problem-solving knowledge. A key task in constructing an expert system is knowledge acquisition: the extraction and formulation of knowledge (facts, concepts, rules) from existing sources, with special attention paid to the experience of experts in the particular problem domain being addressed.