FIPS PUB 77

FEDERAL INFORMATION
PROCESSING STANDARDS PUBLICATION

1980 SEPTEMBER 1

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

GUIDELINE FOR PLANNING AND MANAGEMENT OF DATABASE APPLICATIONS

CATEGORY: SOFTWARE
SUBCATEGORY: DATA MANAGEMENT APPLICATIONS

U.S. DEPARTMENT OF COMMERCE, Philip M. Klutznick, *Secretary*
Jordan J. Baruch, *Assistant Secretary for Productivity, Technology and Innovation*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

Foreword

The Federal Information Processing Standards Publication Series of the National Bureau of Standards is the official publication relating to standards adopted and promulgated under the provisions of Public Law 89-306 (Brooks Act) and under part 6 of Title 15, Code of Federal Regulations. These legislative and executive mandates have given the Secretary of Commerce important responsibilities of improving the utilization and management of computers and automatic data processing in the Federal Government. To carry out the Secretary's responsibilities, the NBS, through its Institute for Computer Sciences and Technology, provides leadership, technical guidance, and coordination of Government efforts in the development of guidelines and standards in these areas.

In June 1979, the Comptroller General of the United States in a report to the Congress entitled "Data Base Management Systems—Without Careful Planning There Can Be Problems" cited the need for technical guidelines to help agencies determine when to use a DBMS and how to evaluate and select DBMS. Within the context of the ongoing effort to develop a family of Federal database system standards, the NBS is pleased to provide these guidelines as an initial step to meet the need for technology planning and selection assistance. Comments are welcomed and should be addressed to the Director, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C. 20234.

James H. Burrows, *Director*
Institute for Computer Sciences
and Technology

Abstract

The Federal Government uses computers principally to process and maintain large collections of data, often called databases. Databases may be separate files on personnel, property, finances, etc., or they may be integrated collections of all such information for a bureau, agency, project, or Federal program. Database users and system administrators face difficult tasks and choices in effectively applying available software technology to their particular agency needs. They often choose a general purpose database management system (DBMS) as the primary software that will enhance application services and improve overall economy. But DBMS usage has special risks because of the technical complexity of these software packages and the scarcity of skilled support personnel, among other factors. This Guideline is a technical primer for Federal managers and applications analysts, to advise them of alternative software capabilities and recommended development practices for database applications. Specific guidelines address applications planning and management, and software selection.

Key words: Computer applications; computer programs; data administration; data processing; data resource management; database management; database standards; Federal Information Processing Standards Publication; file processing; software; software selection.

Nat. Bur. Stand. (U.S.), Fed. Info. Process. Stand. Publ. (FIPS PUB) 77, 50 pages.
(1980)
CODEN:FIPPAT

For sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, Virginia 22161.

Federal Information Processing Standards Publication 77

1980 September 1

ANNOUNCING THE

GUIDELINE FOR PLANNING AND MANAGEMENT OF DATABASE APPLICATIONS

Federal Information Processing Standards Publications are issued by the National Bureau of Standards pursuant to the Federal Property and Administrative Services Act of 1949, as amended, Public Law 89-306 (79 Stat. 1127), Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 Code of Federal Regulations (CFR).

Name of Guideline: Guideline for Planning and Management of Database Applications.

Category of Guideline: Software, Data Management Applications.

Explanation: This Guideline explains alternative software capabilities and recommended development practices for database applications, with specific advice on applications planning and management, and on software selection.

Approving Authority: U.S. Department of Commerce, National Bureau of Standards (Institute for Computer Sciences and Technology).

Maintenance Authority: U.S. Department of Commerce, National Bureau of Standards (Institute for Computer Sciences and Technology).

Applicability: This Guideline is intended as a technical primer or basic reference for Federal managers and applications analysts who are responsible for computer applications and associated software and project decisions. Its use is encouraged for planning and management, but is not mandatory.

Implementation: This Guideline may be used to prepare agency directives or management procedures to implement its recommendations in data management activities. Use of this Guideline should be considered when new or upgraded systems and techniques are contemplated for computer processing of large data collections.

Specifications: Federal Information Processing Standards Publication 77 (FIPS PUB 77), Guideline for Planning and Management of Database Applications (affixed).

Qualifications: Most of this Guideline is pertinent to general data management applications, i.e., any that process numerous similarly defined data aggregates or records. This Guideline also highlights the special problems of using generalized database management system (DBMS) software packages. At present, with no pertinent international, national, or Federal standards, all commercial DBMS are unique. This Guideline describes the three major classes of DBMS within which some similarity arises among commercial products. These classes are the focus of the current standards effort. The generally perceived advantages and difficulties of each class, or data model, are described, but cost and benefit data are not available to permit a government-wide recommendation as to their suitability for specific application needs. This Guideline provides the basic orientation to planning and selection issues for which continuing NBS effort will provide more specific guidance.

Where to Obtain Copies: Copies of this publication are for sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, Virginia 22161. When ordering, refer to Federal Information Processing Standards Publication 77 (FIPS-PUB-77), and title. When microfiche is desired, this should be specified. Payment may be made by check, money order, American Express card, or deposit account.

Federal Information Processing Standards Publication 77

1980 September 1

Specifications for

GUIDELINE FOR PLANNING AND MANAGEMENT OF DATABASE APPLICATIONS

CONTENTS

Figures

1. PURPOSE AND SCOPE

The objectives of this Guideline are to give the reader an appreciation of the possible pitfalls in database projects, and to explain technical management methods which contribute to successful development of database applications. Software selection and the major problems of DBMS usage are emphasized.

1.1 Use

This Guideline is intended for managers and technical personnel who are responsible for computer applications and for associated software and project management decisions. They may be data processing professionals, or managers and senior professionals in other fields who seldom use a computer themselves, but nevertheless are responsible for government functions that require computer services.

Every reader will need to understand the rudiments of computer operation and programming, as well as the common terminology of data processing. A glossary is provided for the principal terms introduced in this Guideline. Selected references for further study of the important topics are given at the end of this Guideline. References are denoted in the text by an author's last name and year of publication in brackets, e.g. [FIFE 77].

This Guideline may be used to prepare agency directives or management procedures to implement its recommendations in data management applications.

1.2 Scope

In this Guideline, software means the data files, programs, and documents that have to be provided for any application of computers. Today, providing software involves greater cost and risk than providing computer equipment, because hardware is mass produced by industry using proven technology, while software still is produced mostly by the craft of individual computer programmers and users. To limit the software burden, application managers increasingly are using industry produced, general purpose software packages, such as a database management system (DBMS), as key components or building blocks of major applications. At present, it is estimated that half of the Federal Government's larger computer installations provide at least one DBMS for use by their customer population. Even more DBMS installations will occur in the near future, because of improving practicality of DBMS for mini- and micro-computers.

This Guideline summarizes a recommended discipline of application management, which consists of technical and management activities, decisions, and controls that are required to purchase, produce, and maintain software throughout the life cycle of a computer service that supports an organization's mission. Application management is very similar to software management [FIFE 77]. It differs because of two areas of special concern and emphasis: the control and management of large archival data collections, in addition to associated computer programs; and, the selection and management of proprietary software packages that serve as major application system components.

For most of this Guideline, database applications are not distinguished from data management applications. The latter are considered as any that process numerous similarly defined data aggregates or records. This encompasses most so-called commercial applications and COBOL programs, for example, and under this definition the data management category is dominant, by frequency and by investment, among all applications. This broad definition is used in order to present widely useful applications guidance and also to review DBMS usage in proper perspective with all other relevant software approaches. Nevertheless, the special problems of using a DBMS are highlighted in this Guideline.

1.3 Importance of Applications Management

Government and business today are completely dependent on computer services, and social and economic activity is geared to the performance of computer systems. Examples are many, such as airline reservation systems or the Social Security Administration systems that distribute over $50 billion annually to about 25 million individuals.

Computer benefits require effective management of computer software. Software is very expensive, and its procurement and development are major concerns for any organization, particularly the Government [COMP 76 through COMP 80b inclusive]. Estimates of Federal Government spending range as high as $5 billion annually on software, and the accumulated investment in current Federal software probably exceeds $25 billion dollars. For instance, cost estimates for the formerly proposed Tax Administration System of the Internal Revenue Service were well over $500 million dollars, and the Air Force has estimated its software costs for command and control systems in the 1980's will be several billion dollars.

No technology, tool, or standard practice now exists that will surely prevent faulty design, logical errors, cost overrun, or late delivery for any software project. Computer applications development can require the coordinated effort of dozens or even hundreds of programmers and analysts. The possibilities for mistakes and poor decisions may be higher than in any other technical field, because of tedious complexity and the lack of standards and common practices. Success or failure is determined more by the technical and management skills of individual personnel than by any other factor. Now, there is a serious and growing shortage of adequately trained data processing professionals.

This Guideline is a primer for Federal personnel who may not have adequate training and previous experience to confidently meet major responsibilities in providing and using database software. Although not a comprehensive handbook of practice, the information and guidelines herein should assist such personnel to anticipate potential difficulties and to plan project activities and technical actions more effectively.

2. SOFTWARE ALTERNATIVES

This section briefly reviews the alternative software approaches available to meet data management requirements, so that the reader may appreciate the basic choices and their characteristics, advantages, and disadvantages. For thorough tutorial explanations, the reader should use textbooks, published papers, and product manuals, such as [MART 76, MART 77] and others included in the References. The following includes strongly favorable or unfavorable indicators that warrant serious consideration by agency personnel in the software selection process. In some cases perhaps, one condition alone may be sufficient for acceptance or rejection of one alternative for an application.

2.1 Traditional Application Systems

Many computer applications today still reflect the approach dating from the late 1950's, when only magnetic tape mass storage, punched card input, and simple programming language compilers were available. This traditional approach involves specially designed programs that provide all the needed application functions, and that are completely user-written, often in assembly or machine language, but more often in COBOL or another high-level programming language. The data records usually have a simple, common format; for example, they all have the same data elements, comprising one file of fixed length records. Sometimes a few distinct files are involved, such as a master file of archival records, an input file of update transactions, and an output file of erroneous or unprocessed transactions. The files are typically stored on tape, ordered in sequence according to some key field (data element). An application program, for example, to produce a report of selected or summarized data, reads the tape to retrieve each desired record, extracts the appropriate fields, and then composes and writes to the printer each line of the report.

The programming staff develops each application system separately, working out program logic and file designs that seem convenient and efficient for the specific processing required. One result of this process is that the application programs "own" the data, since the file and record descriptions are placed within each program and are an integral part of it. Usually the design and access of files are strongly affected by physical or machine dependent factors, such as the amount (in bytes or words) of main memory buffer space to store retrieved records, or the displacement of one field in bytes from the key field.

In brief, the following characteristics typically apply to the traditional approach:

* program logic and data formats are closely interwoven, with a strong likelihood of maintenance or redesign problems if new processing steps or data items are introduced;
* average skill with the chosen programming language is usually sufficient for an acceptable solution;
* coordination with other applications or with the computer installation management seldom is required;
* physical and machine dependent factors affect the design, and may impede future conversion and maintenance of the application system.

The traditional approach still may be highly effective where conditions such as these apply to the application:

* data may be processed in daily or less frequent batches without unacceptable delay;
* very large amounts of data are to be handled, with a need for high efficiency or performance;
* records and files have relatively simple structure, e.g., fixed size and common format for all records of a file;
* update, processing, and reporting functions are limited, predictable, and unlikely to change;
* ample development time is allowed for the software;

* change to different computer equipment is not expected for some time, e.g., a year at least.

For example, traditional file processing remains an efficient way to handle the very high volume, but infrequent record creation and updating found in many activities of the Internal Revenue Service or the Social Security Administration.

However, traditional techniques normally would be difficult or unsuitable where these conditions apply:

* application needs are in flux, with considerable uncertainty as to the important data elements, expected update or processing functions, and expected volumes to be handled;

* rapid access is needed to answer ad hoc questions;

* substantial dedicated programming assistance is unavailable;

* many data elements are common to distinct agency activities, and must be shared for economy, reliability, or other reasons;

* unusually short development time is demanded.

2.2 Support Packages

Hardware and software advances of the past 15 years, particularly large core memory, disk mass storage, and computer time sharing, have made it practical to satisfy needs that cannot be met by the traditional system. Advances as above help most applications, not just data management. But, they have been accompanied by software innovations that specifically support data management:

* operating system functions to create and access nonserial files;

* transaction processing to manage communication networks, messages, and display screen forms, and to do simplified retrieval and update on files;

* generalized report writer and file processing software to produce reports and to do various sort, copy, and extraction functions without complex programming;

* online query processing to retrieve specific data quickly;

* data dictionary systems to catalog the location, description, and usage of many data elements and files.

These capabilities are used with traditional application programs as well as with database management systems, in order to improve accessibility, responsiveness, and management control. But, they do not themselves introduce fundamentally new capability to organize, relate, and retrieve data in logical or application terms, as do database management systems (to be described in sec. 2.3). The following describes the technical features, advantages, and limitations of the above support packages, that are most relevant to their selection as application components.

2.2.1 Operating System/Transaction Processing. The evolution of magnetic disks as an economical and customary mass storage medium also produced a change from serial to random record processing as a common software procedure. Industry-produced operating systems were improved to provide so-called access methods, known by terms such as index sequential and basic direct access. Access methods differ in the way that application records are placed into the physical records, blocks, or tracks on disk storage, and in the type of indexing and chaining used to locate and retrieve individual records. An access method provides primitive, machine-oriented functions by which the application programmer can build application-oriented indexing, storage, and search routines. Thus, although careful low-level design is needed, the limitations of tape file handling (repeated sorting and serial search over all records) can be avoided, and randomly occurring requests to extract or modify individual records can be done rapidly and efficiently.

Transaction processing is an appropriate extension of operating system capability, but often is incorporated within a database management or other application package [MCGE 77]. It involves utility services needed in passing messages between many remote computer terminals and the application software (possibly a DBMS) that will process them. Pertinent services include control and status monitoring of the communication facility, logging and queueing of messages, and generation and processing of displays, especially predetermined screen formats for clerical data entry and validation.

Such capabilities are indispensable for the online, random access approach to data management. Well-known industry products have relieved the computer user of a substantial burden of producing similar software, which

otherwise would have made the modern system economically impractical for many users. Nevertheless, use of these packages does induce dependence on specific hardware and proprietary software techniques. Eventually, application conversion and maintenance difficulties may prove insurmountable unless the using organization has developed some in-house operating system expertise.

2.2.2 Report Writer/File Processing Systems. To create a traditional program to generate a summary report, a programmer must formulate the logic, write the code, and debug the output—a process that could take days. Report writers or file processing packages can conveniently generate reports with a minimum of programming, and similarly can accomplish various routine utility functions on files.

These systems vary in capability, but in general they provide:

* an easily learned language for nonprogramming users to describe what they want done;
* the ability to access files created and maintained by the operating system, language compilers, and application programs;
* the ability to extract data elements or fields from a record, and to change values, count, add, and perform other simple processing;
* the ability to specify precise editing, formatting, titling, etc., to produce a desired printed report;
* the ability to sort, copy, subset, merge, and perform similar utility operations on one to perhaps six different files.

Examples of report writer/file processing packages are described in [NEUM 77].

Compared to traditional programs, report writers offer advantages of easier coding of routine and fairly straightforward functions, and access to files by end users without programmer support. Use of such packages is often concentrated on time-critical, infrequent, or one-of-a-kind requirements, or situations where trained support in conventional programming languages is lacking. A significant disadvantage is that many of these packages deal with the files in physical and machine-dependent terms, forcing users to know the size of fields, records, headers, and other minutiae in order to describe their needs. Also, the record and file formats accepted may be limited to a few types, and so these packages may be unable to process certain application files.

2.2.3 Online Query Processors. Many organizations and government agencies have a widespread need for rapid access to data by those without programming skills, who must quickly answer client and management questions. This need is met by online computer terminals served by query processing packages, which allow the terminal user to formulate questions in a simplified English or graphical type of language.

Available query packages offer different styles of language. The most common is a line-at-a-time command format, such as the following example for locating employees of certain grades:

FIND PERSON-NAME WITH GS = "12" OR "13"

Online query systems are easy to use, modest in cost, and usually efficient for searching small files. But they seldom are available commercially except as part of a larger package, such as a report writer/file processor or a DBMS. Because they facilitate multiple user access, query processors require additional attention to security and access controls to guard data against unauthorized disclosure or change.

2.2.4 Data Dictionary Systems. A data dictionary system (DDS) is a specialized file processing or database software package used to inventory, control, and manage data elements in a uniform manner [LEFK 77, LEON 77, ICST 77, ICST 80, NBS 80b]. Applications managers always need to keep records of data elements used, their formats, and their meaning, if only in a looseleaf notebook. But the complexity of large volume files used today has inspired the development of special packages to automate this function.

In a traditional programming environment, a DDS can help to standardize data definitions and usage, and to locate existing files and processing programs, saving unnecessary or redundant software development. Appropriate use of the DDS leads to improved documentation, produced concurrently during development rather than after the fact. A DDS may provide features that assist a variety of other important applications tasks, such as auditing or data conversion.

A DDS provides a range of benefits to application management, but may itself be a rather complex package, comparable to a DBMS. Its key disadvantage is the additional technical staff resources to install, maintain, and manage the DDS.

2.3 Database Management Systems

The data management tools described above—query systems, report writers, data dictionaries—do not completely fulfill a fundamental goal of computer data management, which is to bring together most of the data elements found in a major enterprise so that they can be readily managed and effectively shared by many different offices, users, and applications. Traditional programming is even less effective toward this goal, because it produces independent computer applications developed at different times by different groups for different purposes, with separate yet often overlapping data files.

Database management systems are accepted by the professional community as an effective, though technically complex avenue to meet common application goals, such as data integration. This has been a major area of software research and product innovation for 15 years or longer [FRY 76], but is still considered an immature technology that is rapidly changing. A DBMS may be beneficial for four major, yet alternative reasons:

* it provides features and preprogrammed functions to organize and process many different types of data in an integrated, space-saving structure called the database;

* it provides for database definition in generic or logical terms, independent of physical storage and machine dependent factors;

* it is general purpose software that facilitates rapid and economical development of many applications;

* it provides a rather complete operational facility, with features for security, error recovery, monitoring, etc.

Briefly stated, a DBMS provides:

* a Data Definition Language (DDL) for describing the database as individual data elements and associated or related groups of elements (e.g., records, sets, or tables), independently of the computer programs which may access, update, or otherwise process the numerous values (occurrences) of these elements. (The database description is called a schema, and the approach of separating data description from programs is to achieve so-called data independence, a condition in which most schema changes will not require changes to be made in application programs.)

* one or more Data Manipulation Languages (DML), such as an online query language or extensions to a common programming language, by which an end user or programmer can select and process database information;

* operational features, such as access control, diagnostics, activity monitoring, transaction logging, etc., for secure and reliable processing in the presence of errors and abnormalities;

* system software, normally invisible to the database user, that interacts with the computer operating system and hardware (particularly magnetic disk mass storage) to provide a reasonably efficient implementation of the database operations on the given machine.

Many DBMS provide major additional features. For example, the capability to define subschema or "views," by which individual programmers may use a portion of the database suitable to their particular processing, redefining data elements, names, and relationships as they choose. A DBMS may provide extensive operational control functions, including transaction processing support, and physical storage allocation and management features. Multiple types of query processors, report generators, and DML interfaces may be offered also, to help a variety of users access the database. Some query languages permit much flexibility in creating subsets and other new data arrangements. Also, they may perform various statistical operations, to help analytical study of data besides predictable database maintenance.

Vendors have developed and marketed dozens of generalized DBMSs and Federal agencies have developed many special systems with similar capabilities. Yet, an unequivocal distinction between DBMSs and other types of software systems is not easily made. Many application systems have limited features akin to a DBMS. Judging a system to be a DBMS is often a matter of degree or a combination of features, rather than the presence or absence of one specific function.

A DBMS, for instance, achieves data independence by separating the database definition or schema from the programs or queries that access the data. This generalization of a successful COBOL concept (DATA DIVISION vs. PROCEDURE DIVISION) insulates users from many details of the database specification. But, a COBOL compiler with a sophisticated time shared operating system would not be considered a DBMS, even if program DATA DIVISIONs were stored in a library and automatically included at compilation. First of all, such a library would not suffice as the single, controlling database definition that is necessary for a large, integrated collection. Second,

COBOL data structures are limited to fixed-length, predetermined records, and this is inadequate compared with a typical DBMS which conserves storage for variable length data and permits unplanned creation of new record types.

One way that DBMSs can be classified is according to their predominant features and techniques for structuring and operating on the database. This predominant capability now commonly is called the data model implemented in the DBMS. The appendix describes the data model for three common types of DBMS: hierarchical, network (or CODASYL type), and relational. This is a broad, rather than an exact classification. All DBMSs are different in significant respects. This, coupled with the important benefits of a DBMS, is the basic motivation leading to DBMS software standards. The major, present uses of each type, and the impracticality of selecting one type as best for any category of application, indicates that a family of alternative standards is necessary, rather than a single standard system that everyone would use.

The advantages of a DBMS indicated above are offset by many, sometimes serious disadvantages. A mature DBMS is a sophisticated system that can overwhelm the technical capacity of a user organization. Vendors often do not provide sufficient or responsive technical support. It may be very difficult to configure the schema or to tune the DBMS properly for adequate performance and efficiency. A DBMS may take so much of the computer resources and time that it seriously interferes with other production work. Thus, the selection and effective application of a DBMS for any purpose is not a casual effort, but a difficult technical and management task that demands careful analysis and planning.

2.4 Distributed Database Management

Distributed database management is the management of geographically dispersed data by two or more DBMSs located on different computers and using telecommunications for the exchange of commands, status messages, and data. This approach is an advanced technological solution for organizations which need computer resource sharing or rapid, centralized coordination of data, but which have an environment of geographically dispersed and rather independent applications. Military command and control is such an example, and another is environmental monitoring and protection.

Distributed database management can be accomplished with various operating modes. One is to have frequent bulk transmissions of data between computer sites. This provides a reasonably current, composite database for access by the users of each computer. A more responsive approach, now the subject of research and advanced development, is to provide for access by any user to any database via a common, network-wide retrieval language [ROTH 77]. This approach requires substantial software innovation to interface the different systems economically, and particularly, to translate accurately among the different database structures and the common one.

Distributed database management has several possible advantages compared to a single, centralized DBMS approach:

* improved economy or quality, since databases are designed and managed by the most concerned or capable parties;
* greater reliability, since the entire system does not fail if one site does;
* greater flexibility in sizing or tailoring computer capability to individual applications, and better response to local requirements.

The likely disadvantages are the loss of economy of scale, the risk of insufficient technical expertise to develop and maintain a distributed system, and the immaturity of the relevant software technology.

3. APPLICATION MANAGEMENT TASKS

Over 70 percent of all software is developed by the users of computers, rather than the manufacturers of computer equipment or the producers of commercial software. This means that application management must be an important responsibility of every computer user organization. This section describes its scope and basic tasks as a framework for presentation of specific guidelines in the next two sections.

3.1 Scope and Objectives

Applications usually are initiated by managers in such areas as payroll, engineering, plant and supply, etc. Usually these managers control the project budget and schedule, and have the best information on prospective benefits and requirements. But they often are unprepared to conduct the design effort or to judge the results and

recommendations of data processing professionals. So software designers sometimes work independently using their own judgment, with little pressure to estimate reliably both their progress and the remaining work or costs. Application management is meant to assure accountability for project decisions and objectives, and to provide agency executives with visible measures of progress toward approved goals. This occurs through defined checkpoints where important milestones are carefully evaluated by the concerned parties, and where users and designers take their appropriate responsibility for the decisions made.

Application management identifies quality controls, standards, planning factors, and experience data which are used to control and monitor project work. This involves technical and managerial considerations, and is intended to assure a feasible effort as well as user acceptable quality for the end result.

Application management helps control costs during development and after the installed software begins operating. Experience has shown that changes continually occur over the useful life of applications, which typically is more than 8 years. As much as 70 percent of software cost occurs in this redesign and maintenance. Application management procedures stress close management of effort toward approved capabilities, making best use of limited resources.

3.2 Participants

Except perhaps for small projects, application management involves coordinated effort by responsible parties in different offices within an agency. Typically, the parties represent three principal areas of concern and expertise: users, data administrators, and programmers. The following briefly distinguishes the customary roles of these participants.

If users all belong to one organizational unit, their interests are represented by their management, and if not, then usually by a designated office or a body such as a steering committee established for an application project. The user representatives have an important responsibility to understand and to evaluate the basic design being created for them. The users must have an approval role in acceptance of requirements, specifications, and delivered results, despite their limited technical knowledge.

An agency, office, or project may have an established data administrator [LEON 78], whose duties, on the one hand, may cover only routine maintenance of certain data resources, or on the other hand, may include authority over data standards, database design, and software procurement. In the first case, the data administrator would have only a limited consulting role in application management, sometimes serving as a supporting staff member of an application project. In the second case, the data administrator is the key person responsible for initiating and performing most application management actions.

Computer program design and implementation, including testing, is a substantial part of any database application. Technical control by a professional software designer is essential for resolving complex design issues and for directing programmers to produce a specified product within cost and schedule targets. DBMS complexity often requires skills comparable to operating system design, which normally is beyond the expertise of a data administrator.

Because the skills and organizational goals of these three parties are so distinct, significant communication and coordination problems can occur. The organizational arrangements for application management must assure that adequate leadership authority for the complete project is vested in one person or body, and that the tasks and actions described hereafter are covered in the working relationships and assigned duties of all parties. The personnel and organizational arrangements to carry out applications management must be an individual agency determination, based upon legal, institutional, and technical factors. Sometimes, the most practical choice is dictated by nontechnical factors such as individual personnel capabilities or agency tradition. The guidelines to follow do not indicate explicitly the party that should perform the actions indicated, although this is often obvious.

3.3 Tasks

An overview of activities and decisions is presented best with regard to the recognized phases of a system's life cycle. Figure 1 illustrates the life cycle definition used in Federal documentation guidelines [NBS 76]. The INITIATION phase involves the assessment of an existing, inadequate system, in order to determine the feasibility of replacing it with an improved system. The four stages, DEFINITION through TESTING, are the DEVELOPMENT phase where new or improved software is being produced or possibly purchased. The OPERATION phase starts with the installation of the new system and includes subsequent maintenance. Eventually, maintenance no longer suffices for needed improvements, and another development cycle is begun for a major replacement. Figures 2a to 2f at the end of this section help to explain each phase and stage in terms of documents prepared or used by application management. [NBS 76, NBS 79] provide additional information on documents.

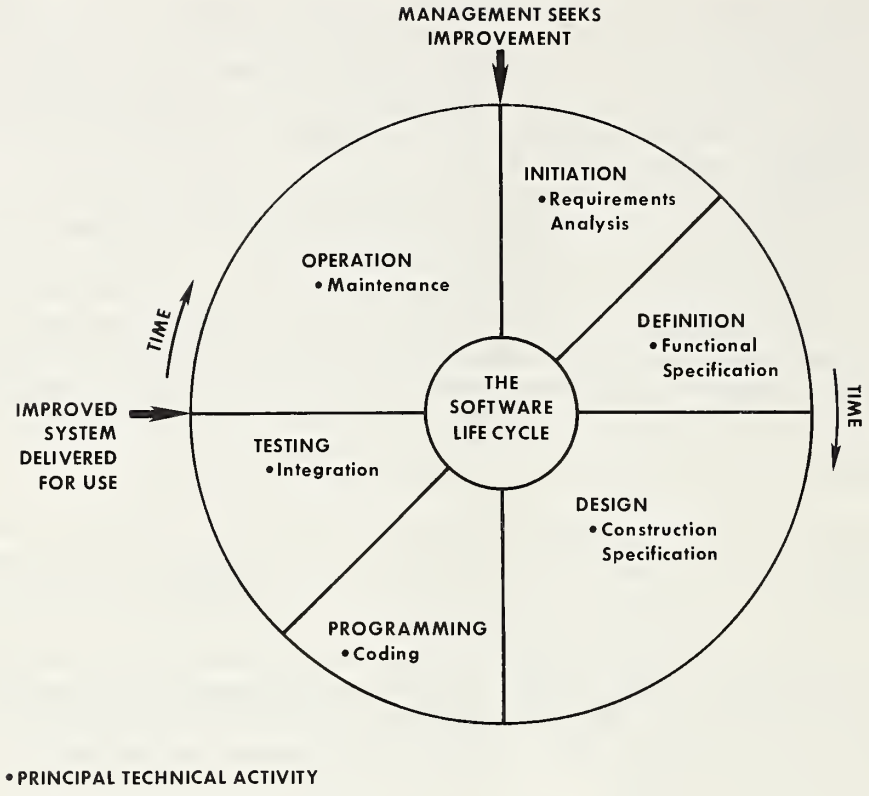


Figure 1. The life cycle of a software product

3.3.1 Initiation. The INITIATION phase begins when management recognizes the need for a new or improved computer service, and through a Project Request, initiates a study to formulate a software solution. During this phase, software planners and application analysts identify all intended users and investigate their work processes. The planners describe the services needed from computer programs, identify broad categories of data and estimate their volume, and then outline alternative software solutions. The alternative approaches should be described in terms that the users will understand. The planners estimate the costs and benefits of each solution. They investigate comparable existing systems to derive cost data, to confirm the practicality of a similar design, and to resolve pitfalls or questionable design issues. This effort produces general specifications of a recommended solution, written to show its scope and feasibility. The INITIATION phase concludes with a Feasibility Study report delineating the recommended software, and also with a Cost/Benefit report containing the supporting analysis. Users must concur with the stated requirements and the recommended software approach. Management then may approve the concept and budget the recommended funds and personnel for acquisition and operation of the planned system.

3.3.2 Definition. The DEFINITION stage defines the Functional and Data Requirements, including performance and quality objectives, so that the software, when built, will meet the users' objectives. Input and output data, processing operations, performance goals, and data collection needs are defined. The specifications cover the externally visible functions and necessary design characteristics, and must be sufficiently thorough for deciding and soliciting any necessary support or product contracts. Once again, they must be validated by the intended application users, to confirm that the proposed system will meet the need.

The DEFINITION stage also provides a Project Plan for managing development, including installation of the system. The Plan includes refined cost projections, and a detailed schedule with intermediate milestones and reviews for evaluating progress. It should define also the working methods of quality control for intermediate and final products. The Plan extends to the installation and initial operation of the system, user training, document preparation and review, acceptance testing, and continued support by contractors. The Project Plan is extremely important because it specifies how the effort will be managed, how problems will be resolved, and how the quality of the final system will be assured.

3.3.3 Design. The DESIGN stage produces thorough design specifications for guiding implementation of the application system and its component application-unique computer programs. The specifications cover decomposition of the application system into subsystems and individual programs, the internal construction of the programs, and the complete description of data elements, files, and databases. The DESIGN effort also covers performance trade-off analyses of algorithms, and definition of interactions between components, such as between a DBMS and application programs. The several specifications are important for resource management and quality control during PROGRAMMING. Procurement of externally produced software components occurs during DESIGN, and usually DESIGN cannot be completed before the selected products have been installed and accepted.

3.3.4 Programming. In the PROGRAMMING stage, application management assures that high quality workmanship is applied in producing code to meet the design specifications. The PROGRAMMING stage often is merged with DESIGN and so may have no distinguishable starting point. PROGRAMMING surely cannot begin before components such as the DDS or DBMS are ready for use. PROGRAMMING involves final technical choices for internal program design and the creation of programming language statements. Application management covers the day-to-day supervision of programmers, the testing of individual programs, and the preparation of pertinent management reports and software documents. Test reports, inspection reports, and program documentation are the tangible quality controls over PROGRAMMING accomplishments.

3.3.5 Testing. The TESTING stage is concerned with verifying that the application-unique programs and purchased software products meet the functional specifications and contain no significant discrepancies or logical errors that would impede their use. TESTING proceeds according to a Test Plan prepared in the DESIGN stage, and focuses on the integration of individual programmer results in an overall working system. Noted discrepancies and errors are documented for corrective action by the programming team or vendor. Gross shortcomings in the original design or functional specifications may be exposed; this should lead to a major project review in order to plan and organize additional definition and design work.

3.3.6 Operation. The OPERATION phase involves periodic redesign and improvement. Application management procedures that are recommended during development can be streamlined to provide similar close control of the reduced effort committed to corrective design work. Thorough specifications, quality workmanship, effective testing, and management review remain highly important in directing technical effort to meet cost and schedule targets.

During DEVELOPMENT, earlier activities may be continued or repeated in order to remedy design defects and improve quality of the final system. All parts of the software need not be at the same stage of development, and the life cycle approach may be used with some flexibility. In particular, proprietary or externally produced software should be put through TESTING and OPERATION before PROGRAMMING is well underway for the application-unique software. This will ensure that these packages can in fact support the application as intended, and will isolate any errors or problems that otherwise would impede testing of the application-unique routines.

DOCUMENT	CONTENT SUMMARY
Project Request	Summary of user need for the development, procurement, or modification of software or other ADP-related services.
Feasibility Study	Analysis of the objectives, requirements, and system concepts; evaluation of alternative approaches for reasonably achieving the objectives; and identification of a proposed approach.
Cost/Benefit Analysis	Adequate cost and benefit information to analyze and evaluate alternative approaches.

Figure 2a. Application management documents—Initiation

Functional Requirements	Summary of the general capability to be developed, stating the requirements and operating environment.
Also includes:	
Quality and Performance Requirements	Desired design attributes and quantitative performance parameters for critical processing operations.
Project Plan	Cost and work breakdown, detailed schedule, and quality assurance and other management methods.
Data Requirements	Description of the required data, with technical information about data collection requirements.

Figure 2b. Application management documents—Definition

DOCUMENT	CONTENT SUMMARY
Design/Coding Standards	General design requirements and coding conventions for programs.
System/Subsystem Specifications	Specification for analysts and programmers of the requirements, operating environment, design characteristics, and program specifications for a system or subsystem.
Program Specification	Specification for programmers of the requirements, operating environment, and design characteristics of a computer program.
Database Specification	Identification, logical characteristics, and physical characteristics of a particular database.
Test Plan	Plan for the testing of software; detailed specifications, descriptions, and procedures for all tests; and test data reduction and evaluation criteria.

Figure 2c. Application management documents—Design

DOCUMENT	CONTENT SUMMARY
Review Reports	Identified discrepancies and recommendations from team reviews.
Unit Test Reviews	Results of unit test for each program.
Inspection Reports	Results of inspection sessions for each subsystem.
Program Listings	Source language statements for each tested program.
Users Manual	Description of the functions performed by the software, written in non-ADP terminology so that the user organization can determine its validity and quality.
Operations Manual	Description, for computer operation personnel, of the software and the operational environment needed to run the software.
Program Maintenance Manual	Description, for the maintenance programmer, of the information necessary to understand the programs, their operating environment, and their maintenance procedures.

Figure 2d. Application management documents—Programming

DOCUMENT	CONTENT SUMMARY
Test Analysis Report	Results and analysis of the scheduled tests.

Figure 2e. Application management documents—Testing

Fault Report	Symptoms and circumstances of detected or suspected failure.
Specification	Functional and design specifications for maintenance and improvement work.
Maintenance Plan	Work breakdown, cost estimate, and schedule for proposed rework.

Figure 2f. Application management documents—Operation

4. GUIDELINES

4.1 Initiation Phase

*Allocate substantial project time and effort to analyze and validate application requirements and to specify or design the computer software.

Nearly 50 percent of software flaws and errors exposed in final testing or initial use of a new system are traceable to shortcomings and mistakes in the early stages of a project, i.e., from INITIATION through DESIGN. Most authorities suggest that added effort in the early stages is the best way to improve the quality of applications. The effort up to the start of PROGRAMMING may reasonably comprise about 40 percent of the total, but working methods must be geared to strong improvements in the validity and detail of the design specifications.

*Use available guidelines, technical reviews, and standardized documents for quality assurance beginning with the INITIATION phase.

Software quality assurance [FIFE 77, BUCK 79], an accepted discipline for large scale systems, emphasizes requirements for maintainability and other often overlooked factors, and also stresses methods to achieve reliable software. Many of its methods also benefit modest applications [BRAN 80], particularly the recommended review methods. The early, formative activity of a project must correlate and evaluate various proposals and data to decide their validity and importance to application objectives. Intensive technical reviews, bringing in distinct expertise and viewpoints, are essential to expose problems or omissions and to decide project scope and priorities. Technology reviews [TSIC 76b, ULLM 80], and especially guidelines that NBS provides on problem areas such as security [NBS 80a], help identify common needs or problems and potential solution techniques. Documentation standards, covering some working documents and data collection forms besides the prominent results [NBS 76, NBS 79], help to make comparisons of evolving specifications and to identify omissions. Computer-aided text editing especially helps to manage the many changes in specifications that occur during recurring reviews.

*Start a Project Record with the INITIATION phase and continue it throughout the application life.

The Project Record should be an open reference to major technical and managerial decisions, and to the current documents and specifications. It should clearly state firm commitments, and should be openly available to all participants.

*Define all user tasks and pertinent performance factors to be supported by the software.

The documentation of the INITIATION phase should characterize the working environment in which the proposed software will operate. This would include a description of user tasks to be supported, the present methods and computer aids for carrying out these tasks, and the present limitations that the proposed system would overcome. The description should be nontechnical and simplified for users' review, so that they may identify omissions or misunderstandings that would affect the proper software solution.

*Maintain continuing interaction among the users and the application designers.

Intended users and their managers are authorities on the working environment, and can provide vital information on the acceptable performance and functions, such as input data characteristics, tolerable response time, etc. A recommended approach is to have selected users serve as members of the project team, not necessarily to perform design work, but to help to decide and to document requirements. Project milestones should include scheduled review sessions as the requirements and alternative solutions are evolved, to obtain user comment and concurrence.

*Formulate and evaluate alternative solutions that differ in risk and cost, but that show innovation in using the computer.

Every application requires some unique software, even when a DBMS or other generalized package is used. But the high cost and risk of in-house software development must constrain the scope of feasible application systems. Meeting needs through commercial computing services and available software packages always should be examined as one alternative, unless expressly forbidden. Innovative approaches beyond routine automation of existing manual systems may yield possibly greater benefits from the computer. For instance, the way in which data records are organized for manual handling may not be best to use in computer storage and retrieval. Query techniques can provide rapid retrieval of the very data that people need, and serve better than requiring people to hunt manually through long computer printouts of all related data.

*Prove out new approaches through pilot software.

Too often, the cause of applications failure is overambition or executive zeal to bring automation benefits to a pedestrian manual operation in one great leap forward. Never undertake to produce in a single project step a system for which there is no precedent. Keep the scope of systems within the bounds of successful project experience. Advance application capabilities in modest steps by developing prototype or pilot systems, which may later be discarded.

*Use consultants and visit other organizations to identify potential problems in building a system and to verify that system objectives are attainable.

Systems that have any possibility of challenging the state of the art must be planned cautiously. Researchers, consultants, and organizations who have pioneered an innovative type of system can help define the major pitfalls and show how to avoid them.

*Expand the planning for new applications to investigate data integration and common software benefits.

Although the scope of a new project may be well known and limited, take the opportunity to look at other requirements, to see how redundant data collection and handling can be reduced and whether new software could bring wider improvements than first imagined. Higher management action often is needed for this step, but the recommendation to do it may be received enthusiastically.

*Systematically evaluate available software of the needed type, from other users and commercial producers, before making a firm decision to develop a new system.

A comparative feature analysis is useful for evaluating available software, and comparing system requirements against currently offered products. Available software, if adequate to the need, can usually be purchased for much less than the cost of producing similar programs. This is especially the case when capabilities akin to a DBMS are required. Even if new development is required, existing software may serve for some components and reduce overall system cost.

*Estimate proposed system costs by several techniques and adopt a conservative projection.

There is no well-formulated cost estimation method that is highly favored and widely used. Various published data exist on programmer production rates [BROO 75], but different variables are involved and some important factors are omitted in one case or another. Experience is certainly an acceptable factor in an overall recommendation. Examine costs of comparable systems and commercial packages for added evidence. Do a detailed cost breakdown of proposed systems by major components and by major technical activities, such as conversion and data standardization. Include operation and development costs, to have a full life cycle investment comparison of alternative systems. Estimate savings from integrating data files and pooling software requirements. Above all, recognize the uncertainty of

any early estimate of cost, and choose a conservatively high, yet reasonable figure. Reevaluate the cost estimate periodically as the development proceeds and the software design evolves.

*Begin training for in-house database expertise.

The advances occurring in small computer and DBMS capabilities may make their use inescapable in the future. Even where applications now are predictable and not amenable to the database approach, a simple DBMS application could arise in some urgent, new requirement. No organizational commitment is necessary to assign one or a few data processing personnel to become knowledgeable in the field and experienced in using these tools in limited, pilot cases.

*Define the system carefully in regard to different types of users.

The user's judgment determines the success or failure of every system. Since major software components and innovations may be needed in some cases, it is paramount that each class of user, reflecting particular skills and goals, is well understood and accommodated in the requirements.

*Have the intended users review and approve the requirements and recommended solution defined in the INITIATION phase.

User concurrence could be as simple as initialing the Project Record for a small project, but may require an official report and approval letter for a large project. To reach their assessment, users should be briefed thoroughly on the cost and benefit analyses, and on the envisioned operation and use of the recommended system.

4.2 Definition

*Resolve uncertain requirements, rather than generalize the design.

If an imposed schedule requires starting DESIGN before all requirements are defined, the unknown requirements must be bounded or limited before further effort proceeds. Be cautious about generalizing the required system capability to handle arbitrary, unknown demands. Generalized systems often involve great risk in development, and may have poorer performance and efficiency than more constrained and specialized designs. Be forceful in investigating needs, in order to resolve uncertainties as early as possible.

*Use analysis, modeling or simulation, and measurement or experience data to establish the performance needed in critical software operations.

Software designers often assume that necessary performance can be attained by "tuning" programs once they are running. On the other hand, early decisions could be so poor as to eliminate any hope of reaching a needed level of performance. Predictive analysis and measurements from existing software should be used to investigate how the system performance depends upon performance of basic steps or operations, such as searching an index. Minimum acceptable performance should be determined for critical operations and then used as validation criteria for the DESIGN and PROGRAMMING stages. Early decisions that constrain the design or implementation approach must show evidence that the necessary performance can still be achieved.

*Include performance and quality objectives with the Functional Requirements that the delivered software must meet.

The specification of quality criteria should begin in the DEFINITION stage in order to have a basis for judging the subsequent design. Use definitions as in [FIFE 77, BOEH 78] as a starting point. Include quantitative performance goals for principal algorithms and processing steps. In the Project Plan, define the technical reviews and the evaluation procedures that will be used to assure that the quality and performance objectives are met throughout the project.

*Develop the Functional Requirements in a structured format, to expedite review and validation.

A free-form narrative approach to documenting requirements will make it difficult, except for very simple applications, to isolate individual functions and performance figures for later confirmation of design. A better approach is first to decompose overall requirements into major functional groups, such as data entry or report generation, or into processing modes, such as online query or transaction auditing. Then describe individual requirements under each group with a separate statement of a function that must be provided, or a performance or design attribute that must be achieved. Each requirement statement should cover the means to confirm its achievement, such as a test or observation.

331-191 O - 81 - 2

*Use data flow diagrams and a Data Dictionary System to begin to catalog and standardize data elements and entities.

A data flow diagram [GANE 79] provides an easily understood picture of application processing steps, to complement the Functional Requirements and to support the Data Requirements preparation. The detail need not always go down to individual data elements. Future effort is saved, and a more systematic analysis effort results, if the diagram content is captured in a Data Dictionary.

*Establish a Data Dictionary System (DDS) to support all future applications.

Management control, standardization, and future conversion are improved by using a Data Dictionary. Small applications should be aggregated, under appropriate management arrangements, to achieve a feasible DDS application environment. The DDS is a key tool for all data management applications, whether or not a DBMS is involved [ICST 80].

*Conclude the DEFINITION stage only when specifications are complete and approved, incorporating changes and refinements emerging from user reviews.

Never proceed into detailed design when ambiguities or known errors exist in the requirements specifications. Minor disagreements are more likely to cause fundamental redesign than easily rendered adjustments.

*Formulate concrete, measurable milestones to monitor progress, and to correct problems as early as possible.

It is impossible to exercise control and take corrective action without concrete evidence of where problems occur. Project milestones must be events and accomplishments that are directly observable by the application manager, such as the receipt of a specified document, the successful operation of a given program, or the conclusion of a scheduled review meeting. Progress during DESIGN and PROGRAMMING should be measured by accomplished reviews and specifications, and not by estimates of percent of code completed or data defined. Recognize especially that the cost of correcting an error or adding capability to a system increases many fold as a project proceeds. By the time system testing begins, errors are over 10 times more costly to fix than if they had been discovered during design. Additional time given to improving specifications and to critizing them thoroughly is worthwhile in reducing later project costs.

*Plan at least 10 percent of total project effort for management requirements and technical reviews.

When the needed effort has been underestimated, which too often is the case, careful reviews and management analysis may be thrown aside to put all effort into design and programming. Obviously this will work to the detriment of quality objectives, and probably will result in even more time required for testing and rework later. Make sure that project estimates include the time and effort for detailed review of work as it progresses, and for continuing review of project schedules and resource expenditures. Insist on using this time as intended, so that project status and problem sources can always be accurately defined.

*Plan contracts for design and production only after the detailed Functional Requirements, Data Requirements, and Project Plan have been prepared and approved.

Contractors cannot be selected or effectively monitored without the specific objectives and quality assurance provisions embodied in these documents. They also provide the first practical basis for decomposing the development effort into work packages that different contractors might undertake. Contract requirements particularly should include documentation standards that will aid in tracing and correlating proposals or designs with the previously approved requirements.

*Plan a modular design that will use commercial packages insofar as practical to provide generalized functions.

Modular design, which breaks a system into distinct subsystems with standardized interfaces, will aid future maintainability and enhancement. In particular, commercial packages can be integrated into the overall system, reducing the amount of application-unique development, or else providing special tools that would be too costly to develop in-house.

4.3 Design

*Plan improved programmer work methods to raise the quality of programs as they are produced.

About 40 percent of software errors are traceable to misinterpretations of specifications, errors in designing to them, or mistakes in writing program statements. Recent research to improve software management has concentrated on these problems. A number of new techniques, such as structured programming and automated quality control, are beginning to come into prevalent practice, but much more must be done. Continuing training programs, special in-house workshops, and experimental projects are good avenues for stimulating the computer staff toward improving their working practices and technical knowledge.

*Prepare explicit quality design and programming standards for DESIGN and PROGRAMMING at the beginning of the DESIGN stage.

The Functional Requirements should include definitions of mandatory quality objectives that serve as a starting point toward the Design/Coding Standards. Using these, a list of design attributes can be specified, extending in detail to conventions on the use of programming language features [LYON 80] and constraints for individual program design. This expansion of quality specifications will provide the Design/Coding Standards to be used in the project for guiding overall design of every program. For example, the quality standards may define the extent of diagnostic messages to be provided, the types of data errors to be detected by validation algorithms, the manner of isolating particular processes subject to future modification, etc.

*Include database schema and subschema standards in the Design/Coding Standards.

Where such standards have not been provided centrally, say by the data processing group which maintains the DBMS, they must be developed for each project, accounting for any overall constraints, such as database limits, that are unchangeable. These standards must reflect the limitations on mapping from schema to subschema, and quality considerations that help database integrity and schema/subschema maintenance in the future.

*Develop the system and subsystem designs as a hierarchy of processes, specifying the input and output of each process, and the data flow interconnections.

A hierarchy is an effective means to decompose functional requirements into successively more detailed operations. The approach is well matched to what is called top-down design for software, which proceeds by successive refinements of general functions, until arriving at the component modules or programs that must be implemented to build the system. Documentation using this design approach is very readable, for it portrays the overall system operation in an easily followed graphical way. Selected references [GANE 79, YOUR 75a, IBM 74b] describe the technique.

*Select proprietary packages early in DESIGN.

The design specifications for application-unique programs cannot be completed without knowing exactly their characteristics, functions, and interfaces with other programs. Be sure that the Project Plan takes account, in the project schedule and labor distribution, of the time required for procurement and delivery. Do not make further design effort or commitments once procurement delay or uncertainties leave open various incompatible candidates.

*Use benchmarks to give confidence of adequate DBMS performance.

Although costly and difficult, benchmarking is the only accepted approach to give some assurance that a chosen system will be able to accomplish the needed production or response time.

*Install and accept the proprietary software, and train programmers on a pilot application, before completing the design specifications.

The planned design may prove inadequate or inappropriate once the purchased system is understood through hands-on use. A pilot application, which will exercise most of the needed functions in a systematic way as well as give more definite performance indications, is a good step to gain evaluation data for a better ultimate design.

*Complete the database schema and program specifications before beginning to code.

It is vital for program and file or database designs to be completely worked out before programming, data entry or data conversion, and similar tasks begin, so that effort is not based on erroneous design notions and so that there is a complete guide to assure quality work.

*Obtain an approved, formal test plan for the system before the DESIGN stage is completed.

The Test Plan should be started in the DEFINITION stage, for the Project Plan must account for the effort required for the testing strategy to be used. Some details on the number of tests, test inputs, etc., cannot be specified until the design is nearly complete. The Test Plan describes the TESTING stage effort and the testing schedule, and defines the standard test set, i.e., the complete range of test cases that will be repeatedly used to assure reliability of the software. This standard test set must be completed, and correct expected output known, by the start of TESTING. The depth of the Test Plan indicates the reliability expected. Ideally, the Test Plan is developed by a technical group that is independent of the designers and programmers, and that will require tests of perceived weak points in the design. Users should be represented in test planning, and the final plan should be jointly agreed upon by users and the programming manager.

4.4 Programming

*Decompose the system into programs that are simple and easily understood—usually no more than 100 executable program statements or about one printed page when listed.

The advantages of small programs are many. Their functions are more clearly and simply describable. This helps direct the programmer toward a clear objective. Their brevity improves clarity and speeds review for validity and quality. Progress may be evaluated more accurately when the units of programming work are small and readily understood.

*Define general service programs to reduce duplication in the system components.

In decomposing the system into subsystems and programs, keep track of replicated functions and define programs that provide these functions as services that all programs may use. These service programs usually are those that manipulate standard data structures in a system of programs, for example, entering items in tables, retrieving items, managing storage allocation, etc.

*Use standard high level programming languages.

The use of a vendor-unique programming language, especially machine or assembly language, should be prevented, except for very small portions of code where assembly language may be essential to achieving the performance goals of the software. These exceptional cases should be strictly controlled by explicit approval. Keep a standard high level language version of assembly programs for documentation and maintenance assistance.

*Use structured programming techniques.

Structured programming [BAKE 75, YOUR 75b, IBM 74a] is a concept that encompasses programming team management, design methods, and programming technology. Numerous books and articles are available, describing the design and programming techniques especially. Published results from evaluation projects confirm that increases in programmer productivity and reductions in program errors are achieved, often by factors of two or greater [WALS 77]. Investment in the training and management orientation needed to practice these methods is worthwhile.

*Provide design and programming tools to increase productivity, and issue guidelines for using them.

Tools and designs for unique tools needed in the project should be selected as early as possible in the DEFINITION and DESIGN stages. Tools of possible interest are small packages that complement major database components—for example, routines that might use the data dictionary to evaluate proposed schemas. Early training and hands-on use are needed before these can be applied effectively in DESIGN and PROGRAMMING.

*Conduct weekly reviews of individual programmer's work.

This practice helps all team members to keep informed of the design and production as it proceeds, and may act also as a training opportunity. But primarily it focuses the combined talent of the team on the quality of the team output as a whole. Errors or inferior design can be more quickly exposed than with reviews by a single manager. The

results of each review session should be documented as a guide for individual programmers to improve or correct their work. In structured programming, these review sessions are called "structured walk-throughs," to indicate that programmers lead the team through an explanation of the design and operation of their programs.

*Design and program for quality before performance.

Before programmers make any effort to tune their programs for improved performance, all required quality characteristics should be realized. Necessary comments and program arrangements for clarity, maintainability, modifiability, generality, and all required functions should be completed and confirmed in reviews [KERN 76, VanT 74].

*Conduct inspections of large projects at successive stages of design and production.

In addition to team reviews of each programmer's progress, more formal inspections are recommended for large projects, to examine subsystems, schema or subschema designs, and related components produced by different teams. These inspections should be progress milestones in the Project Plan. The inspection team may include experts outside of the teams, in addition to the team chiefs and the responsible programmers. The inspection team leader should not have been a participant in producing the subsystem under review and must have the authority to inform higher management of negative findings. A team size of about 10 persons maximum is advisable, and separate reviews should be made when design is complete, when code and unit tests are complete, and when first integration tests are complete [FAGA 76].

*Use iterative review techniques for validation.

Validation, also sometimes called verification, means confirmation that a system is reliable, and meets its specifications as well as the requirements of the user environment. Team reviews or inspections should always trace design specifications back to functional requirements, taking note of findings from past reviews. In this way, the logical thread that stems from the original specification is followed to the working software, and successively examined for continuity and possible shortcomings. Reviews should evaluate the suitability of the software for the intended use, and whether the necessary quality is being achieved.

*Perform standard unit tests during the PROGRAMMING stage.

Unit testing determines that the individual programs work in isolation from others and meet the specifications given with the programmer's assignment. Unit testing must be performed according to standard criteria, rather than ad hoc methods contrived by each programmer. Many different testing criteria are conceivable, but most have no direct relationship to possible errors and design flaws, and so provide no information to improve reliability. For example, one could require a specific number of test runs per program. But unless each test uses distinct program logic, no more is gained than from running only one test. The most important consideration is to avoid arbitrary choices by each programmer that do not clearly exercise previously untested functions and logic. The programmer provides test reports for use in review and inspection tasks.

*Put programs under source code control when unit testing is completed.

Completion of unit testing signifies internal project delivery of a program for integration with others in the overall software system. Source code control restricts further program changes to formally approved changes that have been decided from an overall system viewpoint as necessary to meet project goals. Source code control is needed for best use of programming effort in the terminal period of the development schedule.

4.5 Testing

*Expect high error rates in programs when formal tests begin; plan about 40 percent of project effort for integration testing.

Delivered software for large systems typically has errors at a rate of one in every 300 program statements. When TESTING begins, the errors present may be three or more times higher than this figure. Testing is the process of executing programs with representative input data or conditions, for which the correct results are known, to determine whether incorrect results occur. Testing provides information that helps to isolate errors. Debugging is the subsequent work of a programmer to identify errors explicitly and to correct them by changes to program instructions or data.

Careful, disciplined design can reduce the need for extensive testing. But, it seems best to be conservative in project schedules as well as cost estimates, so allow a major portion of project time for testing and design rework.

*Schedule progressive tests to build up to a representative full system test.

Testing should begin with discrete tests of basic system functions that are needed in order to do more complex and realistic tests. Simple input, output, and control functions are examples. Keeping each test restricted to a given function allows the results to identify clearly the problems encountered. Ultimately, the software should be exercised by composite test cases that represent full system operation and expected usage conditions. Composite tests should be conceived to stress the software performance. The limits on such factors as accuracy, acceptable volume of data, or number of concurrent users, should be verified as stated in the specifications. The diagnostic features of the software should be exercised as well with test cases having invalid input. The Test Plan should encompass most, if not all, of the capabilities defined in the Functional Requirements. Reviews of test sessions should consider the general suitability of the system as well as strict adherence to discrete functional specifications. Any discrepancies should be considered for rework, rather than delivering a system that in the judgment of users and designers will not be adequate for the user's purpose.

*Use program analyzers to assure that all program functions have been exercised.

A program analyzer is a computer program that collects data about another program, including data about it as it operates. In particular, an analyzer can determine whether test inputs have caused all the program instructions to be used. This verifies that all capabilities have been exercised, and identifies extraneous code that may have no purpose and should be removed.

*Use fault reporting to manage debugging and testing.

It is good practice to have a formal reporting system by which detected errors and discrepancies are recorded and fully described. These reports will help to confirm that all known errors are fixed before delivery. They also help to trace multiple instances of the same anomalous behavior, so that debugging assignments can be made for related problems and the debugging effort reduced.

*Conduct regression tests after program rework has been done.

The TESTING stage is primarily concerned with integration testing, i.e., tests to determine whether different programs, produced rather independently, will work together correctly. When errors are found and eliminated by debugging, new undetected errors may be introduced. Regression testing means that previously successful tests are rerun to detect any introduced errors. A complete, successful run of the entire standard test set should be accomplished on the software before concluding TESTING and delivery to the customer.

*Plan a shakedown period after delivered software is installed.

A shakedown period may be dictated by hardware alone if a new computer system is involved, but this kind of trial use may also effectively supplement the previously performed integration testing of software. The shakedown scenario should be as close to the actual anticipated usage as costs and procedures permit, but do not depend upon the shakedown operation for any operational service. Use the fault reporting process and records of shakedown run time to measure and evaluate reliability. The shakedown period needed to attain stable and reliable operation may be several months, particularly if hardware installation is extended over this time and considerable user training is being done.

*Plan for a possible database reorganization after the shakedown period.

Allow sufficient time and effort in the Project Plan to reformulate the database design and to reorganize the data before the OPERATION phase commences. The shakedown period may reveal many weaknesses in the initial database formulation, and also will provide reliable performance data for reconfiguring the database for most effective production. If the shakedown database design is unacceptable for the OPERATION phase, a possibly time-consuming reorganization [SOCK 79] may be necessary.

4.6 Operation

*Use similar management techniques during the OPERATION phase as during development.

The programming and design effort available during OPERATION is generally reduced compared with the DEVELOPMENT phase. However, the goals of application management do not change; indeed, maintenance costs may dominate the total life cycle cost of a software product. Maintenance work on software is still a design challenge, but one constrained by the schema, program structure, and database already in use. Unless technically controlled, maintenance may degrade reliability, database integrity, and other qualities that were initially present. Maintenance tends to disorganize and complicate the program structure, increasing the potential for errors and the cost of future changes. Maintenance should be conducted under a tight management regime that includes definition and scheduling of manageable work assignments, preparation and review of specifications, team review of design and code, standard tests, and documentation.

*Keep a fault reporting process to manage software reliability continually.

Fault reports continue to be important as maintenance continues over the years. They are needed for application managers to determine accurately if maintenance work is being accomplished as assigned and if greater testing and rework effort is warranted. Be cautious about proposed improvements that are not strongly supported by users or indicated by problem reports.

*Keep a performance monitoring and evaluation process to assure continuing adequate service.

The schema design, storage allocations, and other configuration parameters of the database application may need periodic adjustment as the size and composition of the database changes during operations. Performance data, accumulated by the DBMS, will indicate the bottlenecks and the high activity areas. Continuing data that can be correlated over time is necessary to diagnose accurately the needed adjustments.

*Evaluate software periodically and plan improvements to avoid obsolescence and to minimize future conversion costs.

The high investment in current software and the high cost of replacement often lead to prolonged use of software designs that may become obsolete rapidly if not upgraded. Software should be evaluated periodically to identify modifications that could eliminate operational limitations. Also, rework to standardize or restructure routines or data structures can increase the feasibility of transferring the software directly to a future computer configuration.

*Use configuration management methods to control the status of all the application software.

Configuration management involves close control of operational software modifications, to assure that continuing service is not adversely affected by changes and that the changes made are sufficiently important to justify their cost within the total maintenance effort available. Proposed changes should be agreed to jointly by the programming group and the users, and scheduled with the appropriate priority as part of the total workload of the programming team. Changes should first be made to a test copy of the software, not the same copy used for everyday service. Thus,testing can proceed without delaying operational service, and only a fully checked out system is delivered for use. Use a Data Dictionary System as the configuration control tool for all applications, keeping a record of prior program and schema versions, test files, etc.

*Periodically review the Project Record and current documentation, to ensure that maintenance results are being recorded.

Maintenance responsibility is sometimes misconstrued as a personal prerogative that may be done with each programmer's style, unless management exerts its authority for quality control to meet the organization's needs. Documentation is a tedious but crucial chore, and it is inexcusable for operational software to be altered with no evidence except in the code itself. As in all phases of software work, management action should be as immediate as possible. The team review process is recommended, for it may remove any appearance of management inquisition. Weekly meetings that include reviews of documentation updates may be sufficient to keep project information current and complete.

5. DBMS REQUIREMENTS AND SELECTION

This section provides basic guidance for selecting software to meet application needs, and for specifying DBMS requirements, once this type of software has been elected.

5.1 Overview of the Software Selection Process

Software selection is an incremental decision process that is accomplished in several steps. For guidance purposes, this section organizes the selection effort into four consecutive decision steps, although in practice these steps may not be so distinct.

As explained in sections 3 and 4, the INITIATION and DEFINITION effort in a project identifies application needs, and analyzes them to establish basic system design requirements. Here broad feasibility considerations and gross cost estimates determine a recommended architecture for the application system. This involves selecting a configuration of software components according to basic types (see sec. 2), determining their general capabilities and design characteristics, and identifying those to be obtained from commercial sources. This step will be called the Feasibility Decision.

Following this initial selection, the next step is to specify in some detail all the functions and characteristics that each component and package must have to perform adequately or better in its assigned role in the overall system. These specifications are the content of the system definition. They cover the functional specifications for all software products to be purchased or otherwise acquired. This step will be called the Requirements Decision.

Following the system definition, competitive solicitations typically are issued for needed proprietary packages. From the resulting proposals, specific products are selected which best fit the stated requirements, and these are ordered. This step will be called the Procurement Decision.

A substantial design effort still must follow the delivery and installation, because much of the application-unique software, that which is unobtainable as ready-made commercial packages, must be designed to operate with the proprietary packages. Also, the proprietary software must be tailored to the application (through configuration choices, parameter settings, or system generation techniques). This step is called the Design Decision.

Briefly summarizing these four decisions, with an example for each:

Feasibility Decision: selecting the type of software for each subsystem. For example, deciding to use a DBMS as the central data storage and access component.

Requirements Decision: selecting the essential functions and characteristics for a given component. For example, deciding to use a hierarchical DBMS with a FORTRAN DML interface.

Procurement Decision: selecting a specific product that meets the stated requirements. For example, deciding to use a computer vendor's bundled DBMS.

Design Decision: selecting configuration parameters and interfaces for proprietary packages. For example, deciding to hold all updates in a transaction file for overnight processing.

The following sections provide guidance for the first two steps.

5.2 Feasibility Decision

The Feasibility Decision is made principally from a cost and benefit consideration of proposed software alternatives that have survived a screening on their basic feasibility for the agency and the application. Feasibility often is determined primarily by nontechnical factors. The following questions illustrate matters that could arise to eliminate some application approach, including possibly a DBMS or even the computer altogether:

Legal—is the proposed system or service lawful?

Security—is there adequate protection for public needs?

Policy—is the approach precluded for management reasons?

Budget—are available funds adequate to consider this approach?

Personnel—are there positions and expertise available?

Schedule—is there enough time for procurement, design, and installation?

Procurement—is there adequate competition with this approach?

Resources—is a specific computer required?

To set up for the decision, a number of feasible technical approaches are first formulated, and then screened on the nontechnical issues such as above. Among technical approaches, traditional application programs, data management support packages, and DBMS deserve equal consideration at the outset of applications planning. A large scale, diversified application system typically will use all these alternatives in combination for an optimum solution of all requirements. Also, when existing applications are incrementally upgraded, i.e., redesigned and converted in parts to new hardware and techniques, the resulting overall system usually comprises a mixed approach.

After clearly unacceptable alternatives have been rejected using the legal, policy, and other criteria that may apply, cost and benefit analyses are prepared for each of the remaining technical proposals. [NBS 79] provides guidance on documenting such analyses. Although costs can usually be estimated with some confidence, benefits often are very difficult to quantify. In fact, intangible benefits, such as enhancing staff competence, improving software maintainability, or providing better documentation and technical control, may be important factors favoring the DBMS approach.

At present, no cost and benefit data support any government-wide recommendation on preferable types of software, including DBMSs or one specific type of DBMS. Surveys indicate that many present DBMS users have not adequately justified their choices [COMP 79a, EEC 80a], and that technical problems and high costs are raising doubts about corporate or agency-wide data integration [INFO 79]. With regard to the basic feasibility of DBMS usage, Federal agencies must strongly consider two factors:

DBMSs and some other generalized support packages are technically complex, and sufficient well-skilled personnel must be provided, perhaps on contract, to support and maintain such software;

DBMS complexity often leads to excessive computer overhead, interference with other production, or inadequate response time, so performance may be a crucial factor in meeting requirements.

5.3 Requirements Decision

Having decided to use a DBMS, an agency must carefully specify it, explicitly stating the very important capabilities, but leaving reasonable latitude for competition among commercial products.

5.3.1 Overview of Specifications. DBMS capabilities and support needs may be grouped in several different, but equally logical ways for a specification. The following, based partly on [CODA 76] and [EEC 80c], identifies the major parts of a specification, defining the scope of each capability or part.

Database Definition. This part covers requirements on data element names and characteristics; data structures and relationships; Data Definition Language; types of data; operational aids, as for editing and searching the schema; etc.

Data Manipulation. This covers query languages; report formatting statements; common programming language interfaces for data access and processing; subschema database description language, if any; etc.

System and Integrity Control. This covers storage allocation and management; access control; transaction logging; backup and recovery; data validation; file dumping and data conversion; performance monitoring; usage monitoring and accounting; etc.

Performance, Quality, and Other Requirements. This covers quantitative performance targets and benchmark testing; applicable standards; pertinent hardware constraints, such as available memory, multiple system compatibility, etc.

Support. This covers installation; user training; documentation; continued technical assistance in database design and system tuning; design work for enhancements; future maintenance; etc.

Writing thorough specifications is a difficult task, and the needed assistance is being addressed in the Federal standards effort. For the immediate future, agencies may find assistance in specifications used by prior Federal DBMS purchasers and in preliminary documents emerging from national DBMS standardization activities. It is most important that all of an agency's data management concerns and requirements are considered in formulating specifications. Requirements should not be compromised or modified seriously in order to use existing specifications as an expedient.

5.3.2 Alternative Roles of the DBMS Component. Section 2.3 explains that a DBMS may be beneficial for several alternative purposes. Each of these indicates a distinctly different role that the DBMS may have as an application system component. In deciding to employ a DBMS, the manager or analyst must recognize which role has been chosen, and use this for guiding subsequent specification and procurement actions. For each role, certain DBMS

331-191 O - 81 - 3

functions and capabilities are most important, and must be defined most carefully. Less important capabilities then are open for competition among vendor proposals.

Logical Access Method. For distinct applications that do not require online query, a DBMS may facilitate traditional programming by providing, in effect, a higher level access method (less machine dependent) than available in the vendor operating system. The significant capabilities to specify well would be the DML for the programming language to be used, and compatibility requirements on the DBMS data structures so that they are well suited for the expected program designs. For security and applications control, a subschema data definition language may be needed.

Application Productivity Aid. Where many end users must be served and adequate programming assistance is unavailable, a DBMS may be chosen so that users can set up their files and format queries and reports themselves. Here simplified data definition, data structures, and query language are the important parts of the DBMS specification.

Data Integration Facility. Where many applications have common data, a DBMS may be chosen to reduce redundant storage and to provide more flexibility to obtain new data correlations and to solve unforeseen problems. The DBMS data model and data conversion functions would be the most important parts of the specification.

Comprehensive Application Facility. In its ultimate application, a DBMS is the primary or complete environment for computer utilization. Here it must provide all the facilities suggested by previous roles, and also must have the range of operational control and recovery functions that maintain reliable, secure service for many users and purposes. A DBMS specification here would stress access control, recovery, diagnostic capability, performance and resource management features, in addition to the data model, multiple languages, and other features implied above.

5.3.3 Selection of Data Model. Every role for a DBMS draws attention to its data model. The data model may control satisfaction of important requirements such as ease of use, data independence, flexibility, storage conservation, and performance. To define or to evaluate a data model, both the data structures and the operations on the structures must be considered. It usually is difficult to compare data models and to distinguish fundamental from superficial features. Specifying only one data model as acceptable could be tantamount to selecting a particular DBMS, because as more DBMS features are considered as essential to the model, each DBMS appears to have a unique data model.

Nevertheless, the appendix shows that it is possible, though difficult, to identify classes of DBMS according to the data model. This section discusses advantages and disadvantages of each of the three models defined in the appendix, to aid selection decisions. These three are the major types found in commercial packages, but the following discussion does not compare or evaluate the implementation or other operational features of products. [MICH 76, OLLE 78, TAYL 76, TSIC 76a, and ULLM 80] have additional comparisons and discussion of models.

Hierarchical Model. The hierarchical model is represented in the majority of DBMS products sold and installed today, granting that most so-called flat file systems (such as certain report writer/file processor packages) have a degenerate hierarchy. Hierarchical data structures are suitable for predetermined data relationships, so frequently found, that have a superior-inferior connotation, as parent to child, manager to subordinate, whole to part, etc. Despite this naturalism, this model requires the user to understand fairly complex arrangements when a large, diverse database is assembled with it. Depending on the DBMS implementation, this model can be efficient in saving storage and in processing high volume, routine transactions while accessing records one at a time. It has proven also to be an effective model for query techniques that operate on sets of records.

Network Model. This model provides somewhat more general structures than the hierarchical (which may be taken as a degenerate network), for relating diversified data with concern for saving storage. The resulting database may be very complex, and the user, normally a programmer, must track carefully the current reference position among the data occurrences. For this reason, the network structure is said to induce a navigational approach in processing transactions. The network model is capable of high efficiency in performance and storage use. Query facilities, although available, are less developed for this model than for the other two.

Relational Model. The relational model is widely accepted as the most suitable for end users because its basic formulation is easily understood, and its tabular data structure is obvious and appealing to laymen. Some of its terminology and operations may seem abstract and disassociated from historical computing technique, but with training and use they soon become as familiar as other special terms encountered in the DBMS field. Query language innovations are most pronounced for this model over others. Its prominent disadvantages at this time are the paucity of commercial products implementing it, and their immaturity, which may account for a reputation of questionable efficiency or performance of relational DBMSs.

5.4 Procurement and Design Decisions

The third selection step, which principally involves procurement considerations, is not considered in this Guideline. This step begins when the Requirements Decision has provided a functional specification for a needed DBMS. It includes determination of the acceptable type of contract, formulation of various legal clauses, completion of procurement forms, and other nontechnical actions. [CODA 76] and [EEC 80c] discuss the activity in general, with advice on evaluation teams and procedures. For authoritative guidance on procurement and source selection procedures, Federal personnel should consult their agency ADP procurement office and the General Services Administration.

The last selection decision, Design, is beyond the scope of this Guideline. It involves primarily operational factors, although these are very important since they affect database integrity, system security, auditing, etc. Guidance in detailed application design must account for the type of DBMS and its specific features in the operational control area, and is best provided in a separate guideline.

APPENDIX: DATA MODEL DESCRIPTIONS

1. CONCEPTS AND DEFINITIONS

This appendix describes common approaches to database management. It attempts to cover fundamental characteristics of major classes of DBMSs without becoming bogged down in unnecessary detail. The intended readers are experienced ADP practitioners and managers with a general, highly practical understanding of database concepts and packages. In this section we introduce underlying concepts, define terms, and present conventions that are used throughout the appendix.

1.1 Describing DBMS Characteristics

Today—without any international, national or Federal database management standards—virtually every commercial as well as research DBMS product is unique. Furthermore, extant database management systems are described primarily by reference documents that are sometimes incomplete: the products themselves provide the ultimate specifications. It is difficult, therefore, to characterize classes of these nonhomogeneous and undocumented database management systems. This appendix describes DBMS features by considering the underlying data models upon which some implementations are based.

1.1.1 Definition of Data Model. A data model describes the essential characteristics of an approach to database management without necessarily specifying details of language or implementation. At a minimum, a data model description must specify the logical data structures and the operations on these structures that are supported by the data model.

There is no consensus regarding the definition of data model. Many older textbooks and articles discuss data models only in terms of structures, without considering operations. Notwithstanding the lack of universal understanding of the data model concept, the trend today is toward acceptance of the definition presented loosely above. Research at NBS and elsewhere focuses on developing precise formalisms for specifying data models. Regardless of the outcome of these studies, the following sections demonstrate that the data model concept is a useful pedagogical tool for describing essential features of various approaches to database management.

1.1.2 DBMS Selection. The data model perspective is useful for matching DBMS capabilities to application requirements. The difficulty of selecting a database management system is exacerbated by the current state of the technology and the conditions of the marketplace. Selecting a system would be relatively easy if one database management product or approach were always superior to others, if all DBMS applications required the same database management features, or if all DBMSs were available on every hardware configuration. But with a number of unique DBMS products and a myriad of application requirements for database management support, there is no single best approach. The selection problem then becomes one of matching specific DBMS characteristics to specific application requirements. From a data model perspective, one looks at the essential characteristics underlying a DBMS implementation, and then matches these to specific structural and operational application requirements.

1.2 Human Roles and Interface Languages

Humans interact with database management systems at different levels and in a number of different roles. These interactions are facilitated by interface languages for specific roles and levels. Figure 3 is a tabular listing of interface levels, human roles, and languages for using the DBMS.

1.2.1 Interface Levels. As with programming languages, we differentiate among levels of interaction with DBMS software depending on the amount of knowledge of specific system (DBMS and hardware) characteristics required. High-level interfaces require little knowledge of the database management system or the environment in which it resides. At the highest level, the DBMS and hardware would be virtually transparent to the user. Lower level interfaces are more closely tied to specific characteristics and capabilities of the DBMS and its hardware environment. Interface level is a relative concept; levels are neither clearly defined nor easily differentiated. For discussion purposes only, we have identified human roles and DBMS languages at three different levels:

a. *High*—requires specific application knowledge, but little of the DBMS or its environment.
b. *Intermediate*—requires specific knowledge of the DBMS, but not of the hardware/software environment.
c. *Low*—requires specific knowledge of the DBMS and its environment.

INTERFACE LEVELS	HUMAN ROLES	LANGUAGE INTERFACES
HIGH LEVEL (APPLICATION SPECIFIC)	PARAMETRIC/NAIVE USER — PREDEFINED QUERIES — AD-HOC QUERIES	QUERY LANGUAGE
INTERMEDIATE (DBMS SPECIFIC)	TECHNICAL USER — DATABASE POPULATOR — APPLICATION PROGRAMMER	DML & SDDL
LOW-LEVEL (ENVIRONMENT SPECIFIC)	DATABASE ADMINISTRATOR — DATABASE DEFINER — DATABASE CONFIGURER	DDL & DSDL

Figure 3. Levels of human interaction with database management systems

1.2.2 High-level Interface. Parametric or naive (nontechnical) users require high-level access to DBMS applications. They interact by providing parameters, asking predefined questions, or formulating ad hoc queries. These nontechnical users interact with the DBMS application system via high-level Query Languages (QL).

1.2.3 Intermediate-level Interface. Technical users, especially application programmers, interact at the intermediate level. They perform functions such as populating (loading) the database and developing application programs that access the database. These activities are supported by database access languages that are invoked from within programs written in a general purpose programming language such as COBOL or FORTRAN. The two most important database access languages are the Subschema Data Definition Language (SDDL), and the Data Manipulation Language (DML). These languages allow programmers to declare the subset of the database that they will reference, and to manipulate (store, retrieve, update, etc.) the database, respectively.

1.2.4 Low-level Interface. The database administration function requires low-level interaction between data processing personnel and the DBMS. The database administrator is charged with tasks such as defining application databases and assuring their security and integrity, and mapping DBMS structures onto physical media. Interface languages supporting these two functions are Data Definition Languages (DDL) and Data Storage Description Languages (DSDL), respectively.

1.3 Sample Database

We use a sample database throughout the appendix. As discussed above, different data models are appropriate for different applications; no single database can exhaustively demonstrate the features, strengths as well as limitations, of diverse data models. Nevertheless, we use a single core database for consistency and ease of understanding. In each section, the database is augmented as necessary to bring out the salient characteristics of the data model covered therein.

1.3.1 Schema. The sample personnel database includes three types of data: employee data, education data, and experience data. Figure 4 graphically portrays these three record types as rectangles. The names for record types and for data fields associated with record types are noted inside the rectangles; e.g., an employee record contains an employee number, a name, and a birth date. Relationships among record types are depicted by lines. For a single employee record, there may be one or more education record(s), and one or more experience record(s). The figure is a schema or logical data structure diagram; it specifies record types and logical relationships, but not specific record occurrences in the populated database.

1.3.2 Occurrences. Figure 5 includes specific occurrences for all the records and data fields in the populated database. Note that associated with the employee record for Jones are three education and two experience record occurrences. Although each data model does so differently, the ability to handle repeating information such as this is an important feature of most database management systems.

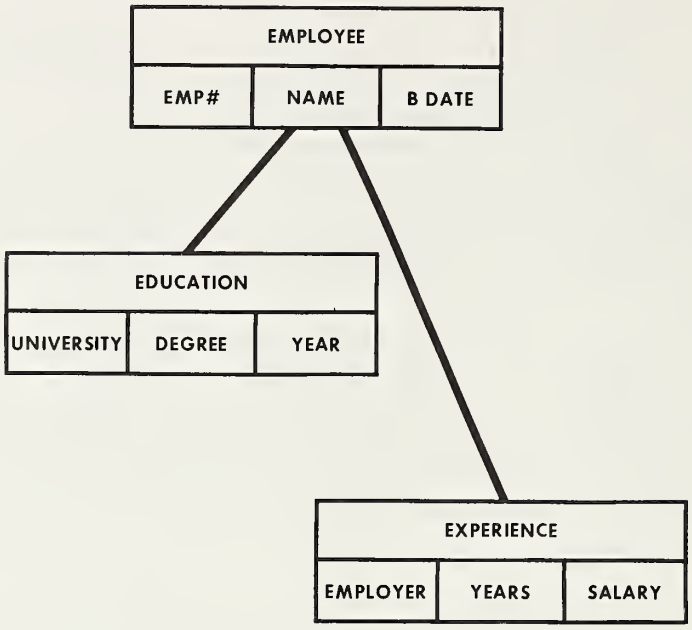


Figure 4. General schema diagram for personnel sample database

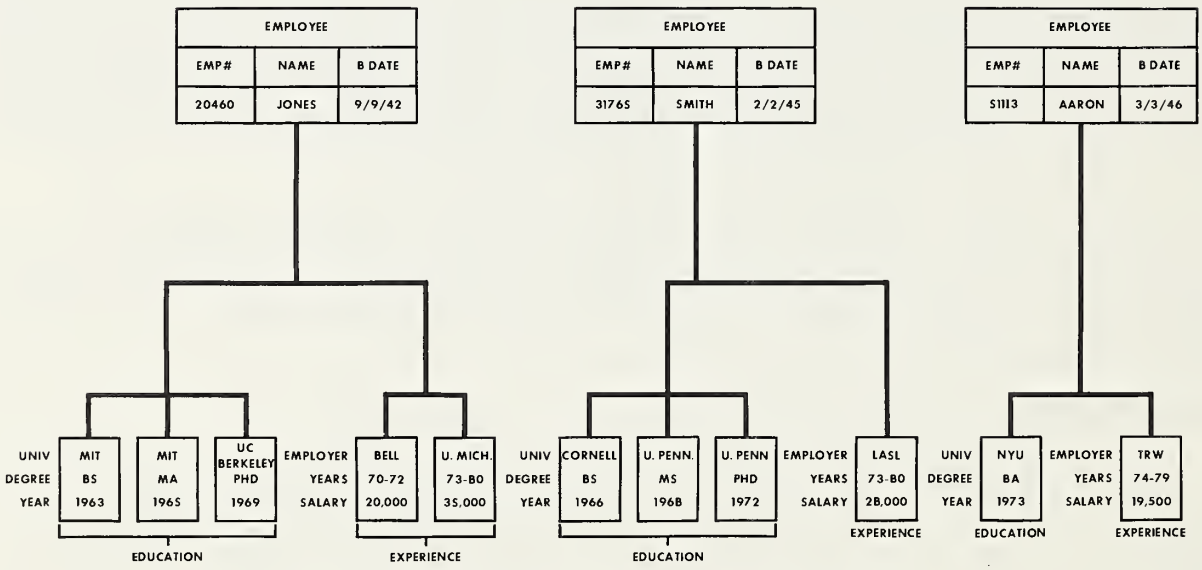


Figure 5. General occurrence diagram for personnel sample database

1.4 Appendix Overview

The rest of this appendix contains sections for each of the three major data models: relational, network, and hierarchical. In each case we briefly identify the model and name some commercial or research products that implement that model. We describe the structures provided by the model, the data definition language for describing those structures, the operations for manipulating the structures, and the user interfaces that use the operations.

2. THE RELATIONAL MODEL

Since about 1970, when E. F. Codd defined the relational model, it has become the major data model for university research. Its strength is its mathematical precision; its weakness has customarily been its performance. People sometimes mistakenly call some commercial packages relational because they support flat files or tabular data; actually, perhaps only two or three relational systems are openly available. Major research systems include System R, INGRES, MRS, and Pascal/R.

2.1 Structures

A relational database does not have a logical structure analogous to those of network or hierarchical databases. In order to discover or display important relationships among data, the user performs operations on fundamental structures called relations. Mathematically, a relation is a set of n-tuples defined over a collection of attributes. In simpler but less precise terms, it is a table of rows (tuples) and columns (attributes) (see fig. 6). Each column has a label corresponding to the name of an attribute; the values in a particular column come from a domain, which is defined either by explicit enumeration or by a range of possible values. For example, a column (attribute) labeled "Color" would contain values like "red," "yellow," and "blue," and one named "Salary" might contain any value between $5,000 and $50,000. One crucial aspect of the relational model is the concept of a unique key; each relation must have a column or combination of columns defined as the key for the relation. The relation cannot contain two tuples with the same value (or combination of values) in the key columns.

EMPLOYEE

EMP#	NAME	B DATE
20460	JONES	9/9/42
31765	SMITH	2/2/45
51113	AARON	3/3/46

EDUCATION

EMP#	UNIVERSITY	DEGREE	YEAR
20460	MIT	BS	1963
20460	MIT	MA	1965
20460	UC BERKELEY	PH.D.	1969
31765	CORNELL	BS	1966
31765	U. PENN.	MS	1968
31765	U. PENN.	PH.D.	1972
51113	NYU	BA	1973

EXPERIENCE

EMP#	EMPLOYER	YEARS	SALARY
20460	BELL	70-72	$20,000
20460	U. MICH.	73-80	$35,000
31765	LASL	73-80	$28,000
51113	TRW	74-79	$19,500

Figure 6. Relational version of personnel sample database

2.2 Data Definition Language

One problem with theoretical formulations for the relational model is the omission of a data definition language (DDL). However, most implementors have found it necessary to provide these facilities in one form or another. Figure 7 shows a hypothetical DDL for our sample database.

The DDL first gives the name of the domain and the form of the values from the domain—e.g., whether they are integer or character. It next tells the domains for all attributes that do not share the domain name. In its final section this sample DDL defines each relation in the database by listing its attributes and its single or multiple keys. In other cases a data definition might also include the definition of views: descriptions of how the data appears to certain classes of users.

Domains are:
NUMBER, Integer(5)
NAME, Char(30)
BDATE, Date
UNIVERSITY, Char(30)
DEGREE, Char(5), Values are: BA,BS,MA,MS,PH.D.
YEARS, Char(10)
SALARY, Money(5), Range:5000-50000

Attributes with names different than their domains are:

EMP# from NUMBER
YEAR from NUMBER

Relations are:

EMPLOYEE: (EMP#, NAME, BDATE)
Key is EMP#

EDUCATION: (EMP#, UNIVERSITY, DEGREE, YEAR)
Key is EMP#, DEGREE, YEAR

EXPERIENCE: (EMP#, EMPLOYER, YEARS, SALARY)
Key is EMP#, EMPLOYER, YEARS

Figure 7. Sample relational DDL

2.3 Operations

The relational model offers two classes of operations: the relational algebra and the relational calculus. The former is more procedural than the latter, but both use conditions on values to manipulate one or more existing relations and produce another relation. The relational algebra originally published by Codd [CODD 70] consists of the following operations:

* Select: Select a subset of the tuples (rows) of a relation based on the values of given attributes.

* Project: Select the designated attributes (columns) of a relation and form a new relation having only those attributes. If, after projection, there are any duplicate tuples, eliminate them.

* Join: Combine two relations by consolidating two tuples into one using one or more common attributes.

* Division: In the simplest form, provides a retrieval capability for sets of objects. Since this operation is not widely used and the definition must be given in set-theoretic terms, it will not be discussed here. Details may be found in [CODD 70] or [DATE 77].

* Set operations: The normal set operations: union, intersection, relative complement and symmetric difference.

There is still no consensus on the best collection of operations. Division is seldom referenced; the join operation has now been expanded into a collection of join operations with a number of different options. The project and select operations as well as the set operations remain as a core of operations with others being proposed from time to time.

With the relational calculus, users still have to know the names of the relations and attributes in the database, but they do not have to specify how the system should process those relations to produce the desired result. Instead, they specify the characteristics of the data using a language based on the predicate calculus of logic and having the general form:

SELECT ⟨attributes⟩ FROM ⟨relations⟩ WHERE ⟨conditions⟩

The relational calculus enables the user to construct new relations from existing relations whose tuples satisfy simple or complex conditions given in the "WHERE" clause.

2.4 User Interfaces

Both application programs and interactive users can manipulate a relational database with either the relational algebra or the relational calculus. The following examples illustrate the use of the relational algebra:

* SELECT EXPERIENCE where Salary > $25000 would yield the relation in figure 8(a).

* To find the universities attended by each employee, the user could project EDUCATION over (EMP#, UNIVERSITY) and receive as a response the relation labeled RESULT in figure 8(b).

* To find the names of the employees and the universities each attended, the user might join EMPLOYEE and the RESULT relation in figure 8(b) over EMP#. The result would be the relation in figure 8(c).

While these operations seem straightforward, more complex queries might require a good deal of expertise to put together the right combination of algebraic operations to give the correct answer.

SELECT

SELECT EXPERIENCE
WHERE SALARY>$25,000

EMP#	EMPLOYER	YEARS	SALARY
20460	U. MICH.	73-80	$35,000
31765	LASL	73-80	$28,000

(A)

PROJECT

PROJECT EDUCATION
OVER (EMP#, UNIVERSITY)

RESULT

EMP#	UNIVERSITY
20460	MIT
20460	UC BERKELEY
31765	CORNELL
31765	U. PENN.
51113	NYU

(B)

JOIN

JOIN EMPLOYEE & RESULT
OVER EMP#

EMP#	NAME	B DATE	UNIVERSITY
20460	JONES	9/9/42	MIT
20460	JONES	9/9/42	MIT
31765	SMITH	2/2/45	CORNELL
31765	SMITH	2/2/45	U. PENN.
51113	AARON	3/3/46	NYU

(C)

Figure 8. Results of three operations in relational algebra

One can also run into some difficulties specifying an appropriate query in the relational calculus, but its nonprocedural commands tend to simplify queries. Two well-documented languages implementing the relational calculus are QUEL (for INGRES) and SEQUEL (for System R). Using SEQUEL, the user could write the following statement:

```
SELECT  NAME, UNIVERSITY
FROM    EMPLOYEE, EDUCATION
WHERE   EMPLOYEE.EMP# = EDUCATION.EMP#
```

The result would be the following relation:

NAME	UNIVERSITY
Jones	MIT
Jones	Berkeley
Smith	Cornell
Smith	U. Penn
Aaron	NYU

The relational calculus thus allows the user to state the attributes desired, the relations from which they come, and the condition for selecting them.

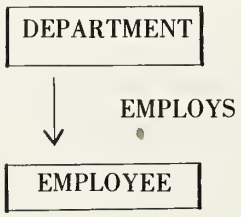
3. THE NETWORK MODEL

The network model offers a variety of data structures for a DBMS. This section focuses on the most important specification of the network model, the one originally proposed by the Data Base Task Group (DBTG) and subsequently developed by this and by other committees of the Conference on Data Systems Languages (CODASYL) [OLLE 78]. Several database management systems—including IDMS, IDS-II, DBMS-10, and DMS-1100—partially implement CODASYL specifications.

3.1 Structures

The major structural unit of the CODASYL model is the record, which consists of zero or more primitive units of data called data items. In the schema the user defines the record types for the database. Each definition of a record type establishes the data items that comprise that type: for example, a record type called NAME might contain data items called FIRST, MIDDLE, MAIDEN, and SURNAME. The user creates and stores a new record as an occurrence of an existing record type. An occurrence of the record type NAME might have "William," "Carlos," and "Williams" for the data items FIRST, MIDDLE, and SURNAME, respectively.

To show relationships among records, the CODASYL model provides for sets (some people prefer to call these structures "DBTG sets" to distinguish them from the mathematical sets used, for example, in the relational model). In the schema the user defines a set type with one owner record type and one or more member record types. The following is a data structure diagram for a CODASYL set. DEPARTMENT is the owner record type, EMPLOYEE is the member record type, and EMPLOYS is the set type:



Each occurrence of an owner record type determines one unique occurrence of the defined set type, and the set occurrence can include as many member record occurrences as the user wishes (including 0 in a null set). Each occurrence of a member record type can belong to *at most* one occurrence of a particular set type. Depending on predefined criteria, the establishment of set membership for specific record occurrences may require explicit program action or occur automatically. Because a DBTG set can have only one owner and any number of members, it

represents a one to many (1 to n) relationship between an owner record and its member records (here and in the remainder of this section, we will use "record" to mean record occurrence. If we mean record type, we will use the word "type"). In figure 9, for example, EMPLED is a set with EMPLOYEE as the owner record type and EDUCATION as the member record type. There are three occurrences of EMPLED, one for each occurrence of EMPLOYEE. In this case every record of the type EDUCATION belongs to an occurrence of the set type EMPLED. The arrows in figure 9 represent logical links binding the owner record to its member records. As the section on operations will show, these links are important features of the CODASYL model.

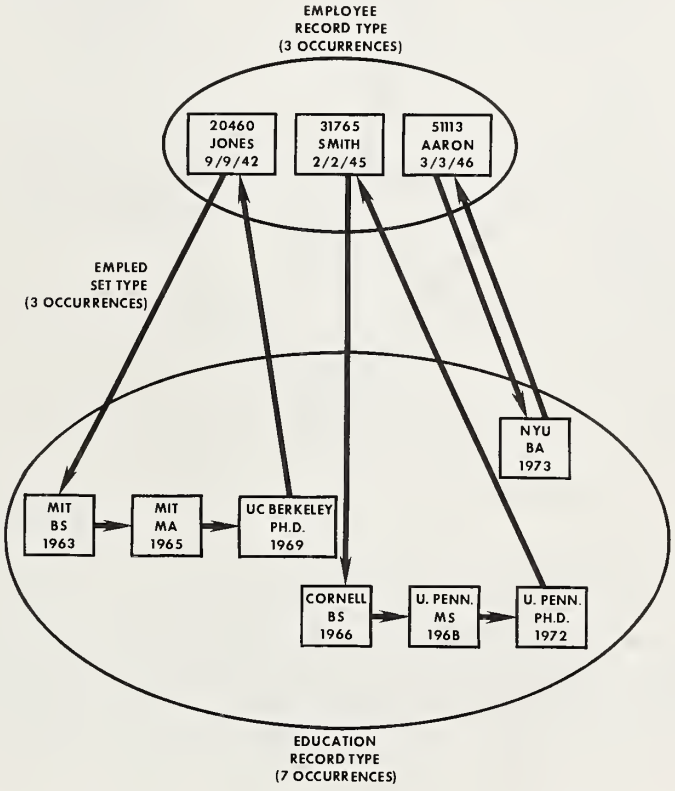


Figure 9. A CODASYL occurrence diagram from the personnel sample database

Because of its simplicity, the sample database does not demonstrate the richness of the CODASYL model. The three diagrams in figure 10 show some of the features that make a network more than a simple hierarchy. In each data structure diagram the arrows represent set types and point from owner record types to member record types. Part (a) shows two set types defined between the same two record types. By definition, *both* are one to many relationships between DEPARTMENT and EMPLOYEE—i.e., no employee belongs to or manages more than one department. There can be EMPLOYEE records, however, that do not belong to a set of either type. In Part (b) the diagram simply reverses the direction of one arrow and changes the set name from ISMANAGEDBY to MANAGES. Now the owner record type is EMPLOYEE, and the member type is DEPARTMENT. Defining the set in this way makes it clear that the data model is not simply a hierarchy with all arrows pointing in one direction. Finally, Part (c) shows a cyclic structure possible only in a network. The data structure contains three record types and three sets. Each arrow represents a 1 to n set from the owner to the member record type. Only one president heads a particular administration, only one administration is in power when a state joins the union, and only one state can legitimately claim a president as a native son.

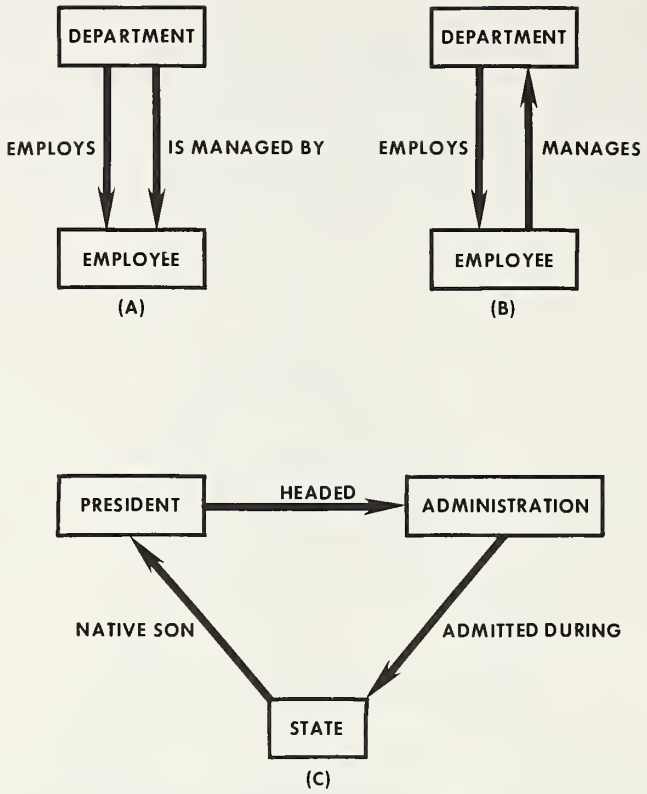


Figure 10. Data structure diagrams for some typical network structures

To make our sample database demonstrate more fully the capabilities of a network, we have added two record types and two sets. Figure 11 is a data structure diagram for this augmented sample database. PROJECT, STAFF, EMPLOYEE, EDUCATION, and EXPERIENCE are the record types. EMPLED and EMPLEXP are sets with EMPLOYEE as the owner and EDUCATION and EXPERIENCE, respectively, as the members. EMPLOYS joins PROJECT and STAFF, and WORKSON joins EMPLOYEE and STAFF. These last two sets exemplify the CODASYL technique for showing a many to many (m to n) relationship between two record types. Since the model prohibits defining such relationships directly, the user can define a third record type to be a member in two different sets, each owned by one of the existing record types. This record could be merely a dummy with no data items, or it could provide useful information like the employee's role on the project and the dates served in that role.

Besides data items, records, and sets, the CODASYL model provides two structures that are necessary to most operations and unique for each program execution, or run unit. The first structure is a series of main memory buffers collectively known as the User Work Area. Operations to store and modify database records write new values from this area into the database, while retrieval operations read values from the database into the User Work Area. Each run unit also maintains currency registers for itself as well as for every record type and set type in the database. These registers are used to store database keys, unique record occurrence identifiers. The current of a record type is the last record of that type referenced by the DBMS; the current of a set type is the last referenced record that is either an owner or a member in an occurrence of that set type; and the current of the run unit is the last referenced record of any type. For example, if the last referenced record in our sample database were the EMPLOYEE record for Jones, the database key of that record would be in the currency registers for EMPLOYEE, EMPLED, EMPLEXP, WORKSON, and the run unit. If we then retrieved a STAFF record for Jones, the currency registers for WORKSON, STAFF, and the run unit would contain the database key of the STAFF record; the currency registers for EMPLED, EMPLEXP, and EMPLOYEE would not change. If the particular STAFF record belonged to a set of the type EMPLOYS, then the currency register for EMPLOYS would also contain the database key of the particular STAFF record occurrence. Most database operations depend on or manipulate currency registers. Some merely insert new keys into appropriate registers, while others perform database access operations as well.

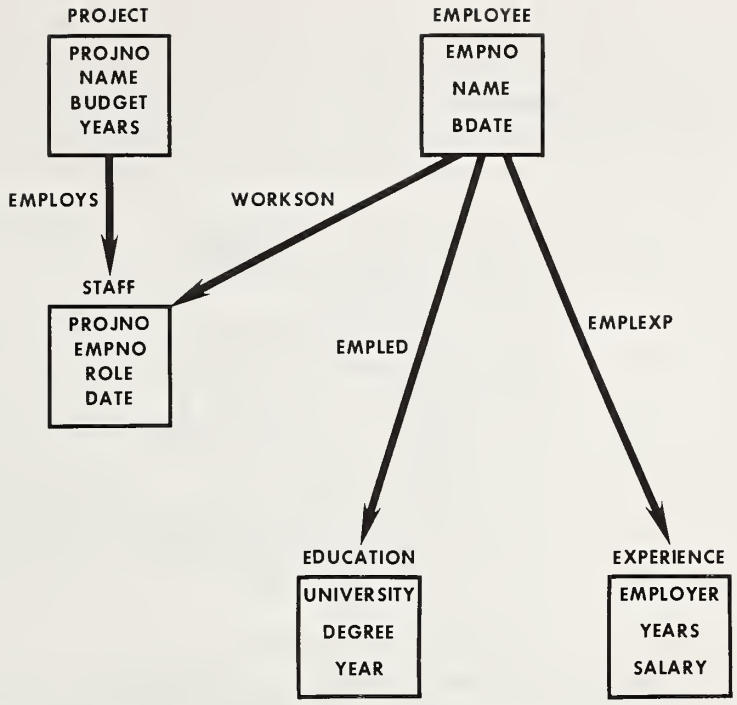


Figure 11. Data structure diagram for augmented sample CODASYL database

3.2 Data Definition Language

To define CODASYL structures, one uses a Data Definition Language (DDL). Figure 12 shows a sample CODASYL DDL with simplified syntax. The SCHEMA statement gives the name of the database. Subsequent statements name the records (e.g., EMPLOYEE), the data items within records (e.g., EMPNO), the sets (e.g., EMPLED), and the owner and member records in sets (e.g., EMPLOYEE and EDUCATION, respectively, in EMPLED). The DDL may also specify such additional characteristics as the possibility of duplicate members of sets or the conditions, if any, that determine set membership.

3.3 Operations

The operations of the CODASYL data model are low-level navigational functions that process a record at a time. Within a host programming language an operation will probably be either a macro or a subroutine call. Below are descriptions of some of the available operations:

* FIND—Find a record occurrence that satisfies a specified criterion. The FIND command has many options; each could be considered a separate operation. For example, one option finds a record if the database key (or DB KEY) is supplied as input. Another option finds the next member of a set. The primary result of the FIND command is establishing the appropriate currency indicators in the currency registers that the CODASYL model provides.

* GET—Read a record occurrence that has been found. This command moves the record itself to the User Work Area.

* STORE—Store a new record occurrence in the database. This command writes a record from the User Work Area into the database and updates currency registers.

* MODIFY—Change values within an existing record occurrence. This command writes modified values from the User Work Area into the database.

* ERASE—Destroy an existing record occurrence.

These operations allow the user to retrieve, insert, delete, and update data.

```
SCHEMA NAME IS PERSONNEL
RECORD NAME IS EMPLOYEE
 02   EMPNO FIXED
 02   NAME CHARACTER 20
 02   BDATE CHARACTER 6
RECORD NAME IS EDUCATION
 02   UNIVERSITY CHARACTER 20
 02   DEGREE CHARACTER 5
 02   YEAR CHARACTER 4
RECORD NAME IS EXPERIENCE
 02   EMPLOYER CHARACTER 20
 02   YEARS CHARACTER 5
 02   SALARY FIXED
RECORD NAME IS PROJECT
 02   PROJNO FIXED
 02   NAME CHARACTER 20
 02   BUDGET FIXED
 02   YEARS CHARACTER 5
RECORD NAME IS STAFF
 02   PROJNO FIXED
 02   EMPNO FIXED
 02   ROLE CHARACTER 20
 02   DATE CHARACTER 13
SET NAME IS EMPLED
 OWNER IS EMPLOYEE
 MEMBER IS EDUCATION
SET NAME IS EMPLEXP
 OWNER IS EMPLOYEE
 MEMBER IS EXPERIENCE
SET NAME IS EMPLOYS
 OWNER IS PROJECT
 MEMBER IS STAFF
SET NAME IS WORKSON
 OWNER IS EMPLOYEE
 MEMBER IS STAFF
```

Figure 12. Simplified CODASYL DDL for the personnel sample database

3.4 User Interfaces

Because the CODASYL model was conceived as a host language system, it favors traditional batch programming techniques. While there are query language facilities, they usually limit access to predefined paths and restrict the structures that can be queried. The following example shows how a host language might use CODASYL operations to print a list of projects and their employees from our sample database:

```
FIND FIRST PROJECT RECORD
[Repeat until there are no more PROJECT records.]
        GET PROJECT
        [Print NAME of PROJECT]
        FIND FIRST STAFF WITHIN EMPLOYS
        [Repeat until there are no more STAFF
         records within the set.]
                FIND OWNER WITHIN WORKSON
                GET EMPLOYEE
                [Print NAME of EMPLOYEE.]
                FIND NEXT STAFF WITHIN EMPLOYS
        FIND NEXT PROJECT
[End.]
```

With appropriate output and iteration statements in the host programming language and appropriate data in our sample database, this series of commands would yield the following list:

```
System Software
        Smith
        Jones
Microprocessors
        Jones
        Smith
        Aaron
Office Automation
        Jones
        Aaron
```

Using variations of the FIND command, the user could also retrieve data satisfying certain conditions; however, retrieval in CODASYL systems is always one record at a time.

4. THE HIERARCHICAL MODEL

The hierarchical model has neither a mathematical basis like that of the relational model nor an existing specification like that of CODASYL. Much of the available information about hierarchical database management comes from two major commercial implementations, IMS and System 2000 [MCGE 77, BLEI 67, HARD 80]. These two systems are very different, and one could easily argue that they represent different data models. We describe the basic tree structure applicable to both models, and then present different data definitions and operations to suggest alternate ways to manipulate tree-structured data.

4.1 Structures

Mathematically, a tree is a connected graph with a root vertex (or node), at least one other node, and no cycles—i.e., there is one and only one path between two nodes. Figure 13 shows a tree with each node designated by a circle and labeled with a letter. Node A is the root of the tree, and its level is 1. Nodes B, I, and J are children of A, and A is their parent. Similarly, nodes C, D, E, and H are siblings. Every node can have as many children as the user wishes, but only one parent (the root, of course, has no parent); nodes that have no children—like C, D, F, G, H, I, K, M, and O—are terminal nodes, or leaves. Because of the restriction of each node to a single parent, the tree structure naturally supports one to many relationships between records. One useful characteristic of a tree is its inherent recursion: any nonterminal node can be considered the root of a subtree.

In a data structure diagram of a tree-structured database, the nodes generally represent collections of data items that we have previously called record types; they are called repeating groups or segments in prominent hierarchical implementations. Figure 14 shows one such diagram for our sample database, now augmented to demonstrate more fully the structures of the hierarchical model. Each node represents a collection of data items. EMPLOYEE is the root, EDUCATION and EXPERIENCE are children of the root, and JOBS is a child of EXPERIENCE. Figure 15 shows one occurrence tree of the sample hierarchical database. Jones is an employee with degrees from MIT and Berkeley, experience with Bell Labs and the University of Michigan, and previous jobs as programmer, analyst, and professor.

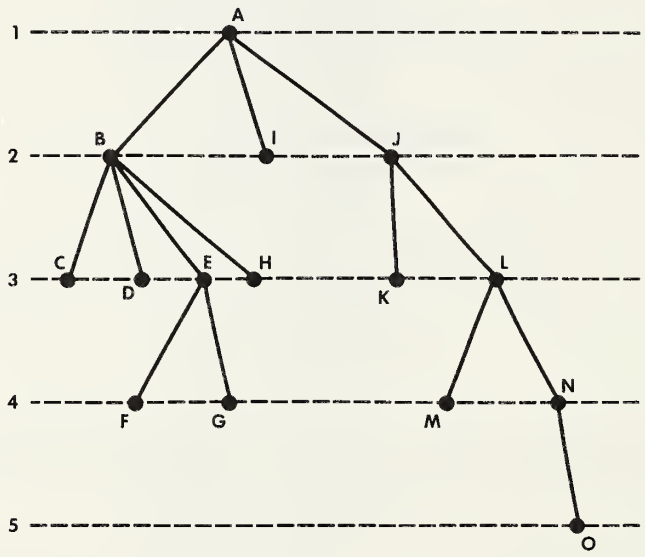


Figure 13. The general form of a mathematical tree

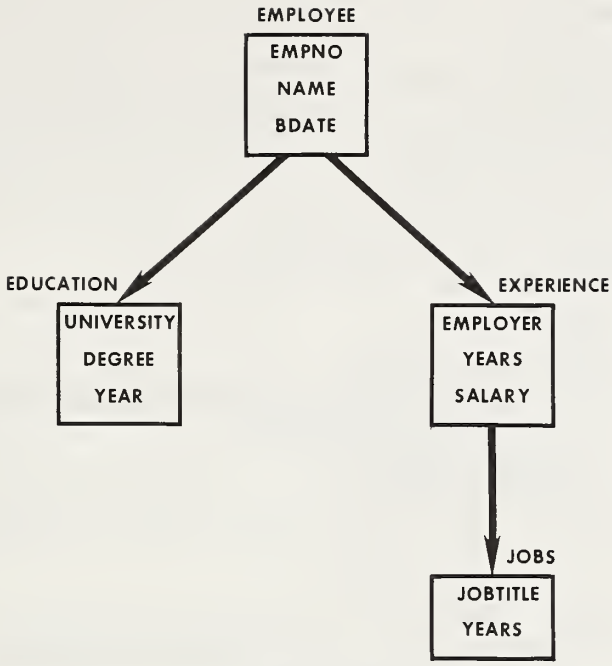


Figure 14. Data structure diagram for augmented sample hierarchical database

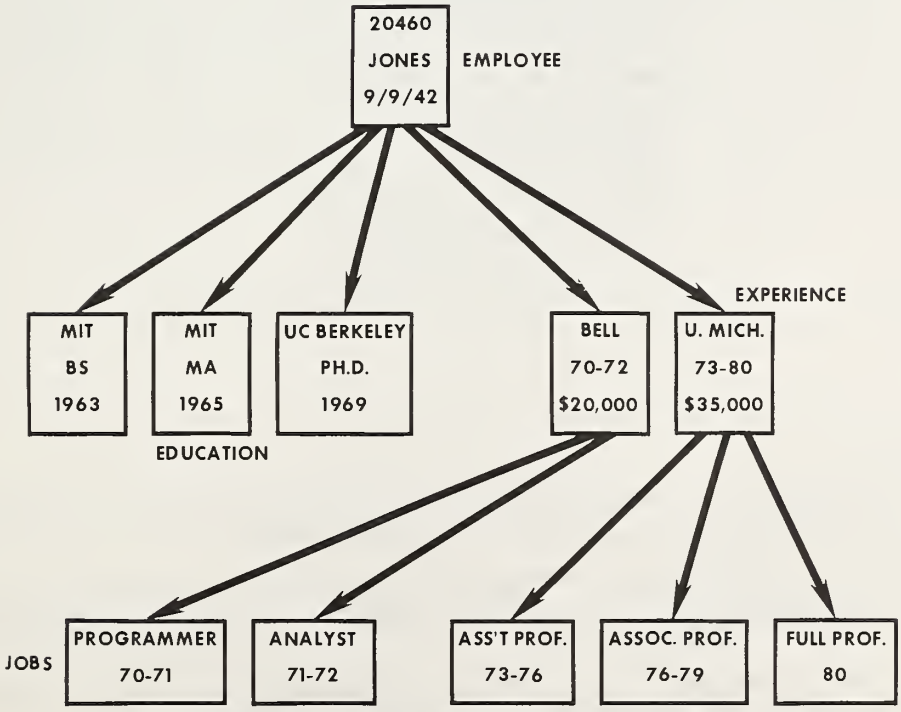


Figure 15. One occurrence tree of augmented sample hierarchical database

4.2 Data Definition Language

Figure 16 shows two possible DDLs for our sample hierarchical database. The first defines each segment type by giving its name, its parent segment type, and its named data items. In some cases the DDL might also specify a particular data item (e.g., EMPNO) as the key for sequencing segments within a segment type. In Part (b) the DDL names data items and repeating groups of data items.

```
DATABASE NAME IS PERSONNEL
SEGMENT NAME IS EMPLOYEE; PARENT IS 0
    EMPNO FIXED
    NAME CHARACTER 20
    BDATE CHARACTER 6
  SEGMENT NAME IS EDUCATION; PARENT IS EMPLOYEE
      UNIVERSITY CHARACTER 20
      DEGREE CHARACTER 5
      YEAR CHARACTER 4
  SEGMENT NAME IS EXPERIENCE; PARENT IS EMPLOYEE
      EMPLOYER CHARACTER 20
      YEARS CHARACTER 5
      SALARY FIXED
    SEGMENT NAME IS JOBS; PARENT IS EXPERIENCE
        JOBTITLE CHARACTER 30
        YEARS CHARACTER 5
```

```
NEW DATABASE IS PERSONNEL
1*   EMPNO (INTEGER)
2*   NAME (NAME)
3*   BDATE (DATE)
4*   EDUCATION (REPEATING GROUP)
     5*   UNIVERSITY (NAME IN 4)
     6*   DEGREE (NAME IN 4)
     7*   YEAR (INTEGER IN 4)
8*   EXPERIENCE (REPEATING GROUP)
     9*   EMPLOYER (NAME IN 8)
    10*   YEARS (NAME IN 8)
    11*   SALARY (MONEY IN 8)
    12*   JOBS (REPEATING GROUP IN 8)
          13*   JOBTITLE (NAME IN 12)
          14*   YEARS (NAME IN 12)
```

Figure 16. Two possible DDL s for sample hierarchical database

4.3 Operations

There are at least two ways to manipulate the data in a tree structure. The first is navigational; the second involves set operations. The following are examples of navigational commands:

* GET: Like the CODASYL FIND command, GET has a number of options. It can find and read a unique segment, the next segment, the parent segment, the next segment within the current parent, and perhaps the root segment.

* HOLD: This command prevents access to the current segment by other users. Together with an appropriate GET command, it allows a user to lock a record while modifying or deleting it.

* INSERT: This command inserts a segment into the database.

* DELETE: After using GET and HOLD to retrieve and lock a segment, the user can delete it with this command.

* REPLACE: After using GET and HOLD to retrieve and lock a segment, the user can replace it with another segment.

Because these commands process one record at a time, they fit in well with traditional batch programming techniques.

In an interactive environment, where the emphasis of data processing activities is probably on retrieval, some hierarchical implementations offer a high-level, general-purpose selection language designed for online retrieval and having the general form:

PRINT ⟨repeating group (or data element or subtree)⟩
WHERE ⟨condition⟩:

This query finds and reads all repeating groups, subtrees, or data items satisfying the given conditions. Unlike the navigational statements, this one employs set operations like union and intersection (for definitions of these operations, see [HARD 80]).

The PRINT WHERE command is especially suitable to interactive queries because it handles large groups of records with a single command.

4.4 User Interfaces

The following example shows how a host programming language might use tree-structured operations to retrieve the name of everyone in an organization who has a Ph. D.:

```
[Begin]
   GET EDUCATION WHERE DEGREE = Ph. D.
   [Repeat until there are no more EDUCATION records.]
      GET PARENT
      [Print NAME]
      GET NEXT EDUCATION WHERE SAME.
[End]
```

This set of commands applied to our sample database would yield the names "Jones" and "Smith."

To ask the same question in a query language, the user might write the following command:

```
PRINT EMPLOYEE.NAME WHERE EMPLOYEE HAS
   EDUCATION.DEGREE = Ph. D.
```

This query would give the same result as the more procedural statements above, although the semantics of set operations are not always so clear as in this example.

GLOSSARY

ACCESS CONTROL
See SECURITY.

ACCESS METHOD
A technique used to obtain data from or to place data onto a mass storage device; usually this refers to operating system software capability provided by mainframe manufacturers.

CODASYL MODEL
A network database model defined by the CODASYL organization. See NETWORK MODEL and [CODA 78].

DATA ADMINISTRATOR
The manager of an organization's data, responsible for the specification, acquisition, and maintenance of the data management software, and for the design, validation, and security of the files or database.

DATABASE
A data collection so organized for computer processing as to reduce duplicative storage and improve the independence of the stored data structure from the processing programs.

DATABASE MANAGEMENT SYSTEM
An integrated set of computer programs that collectively provide all the capabilities required for centralized management and access control of a database that is shared by many users.

DATA DEFINITION
The creation of a schema; that is, the process of identifying and describing both the data elements and their relationships that make up the database structure.

DATA DEFINITION LANGUAGE
The computer processable language used to express a data definition so that the database management system can generate the internal tables, indices, and storage criteria necessary to support database processing.

DATA DICTIONARY SYSTEM
A software tool that records, stores, and processes such information as definitions, descriptions, relationships, and usages about data elements and other data processing entities.

DATA ELEMENT
The smallest named logical unit of data.

DATA INDEPENDENCE
The degree to which application programs are insulated from changes in a database schema or the physical storage arrangement.

DATA MANIPULATION LANGUAGE
A programming language, supported by a database management system, used to access a database.

DATA MODEL
The logical data structures and operations on them provided by a DBMS for effective database processing.

DATA STRUCTURE
A logical relationship among data elements, designed to support specific data manipulation functions. Examples are trees, lists, and tables.

DISTRIBUTED DATABASE
A logical integration of an enterprise's related databases, which are physically stored in a network of geographically dispersed computers.

END USER
A person, at any organizational level, whose occupation requires the use of a database, but does not require knowledge of computers or programming.

ENTITY

A term referring to any concept, object, person, event, or thing that is the subject of stored or collected data.

FIELD

See DATA ELEMENT.

FILE

See [ANSI 77, NBS 77].

HIERARCHICAL MODEL

A data model providing a tree structure for relating data elements, where each node of the tree corresponds to a group of data elements or a record type, and a node may have only one superior node or parent.

INDEXED SEQUENTIAL

A file structure and access method where records may be processed one after another in order of record key, as in SEQUENTIAL PROCESSING, or else a particular record may be accessed directly on the basis of its content, without processing prior records. In the latter case, an index of the file is maintained that relates values of each indexed field or data element with the physical position of every record that has each value.

INTEGRITY

The condition of a database in which the data is accurate and consistent.

LOAD

To insert data values into a database that previously contained no occurrences of data.

LOGICAL

Referring to a view or description of data that does not depend upon physical storage or computer system characteristics.

NETWORK MODEL

A data model providing data relationships based on records and the grouping of records into so-called sets in which one record is designated as the set owner (see SET).

PHYSICAL

Referring to the representation and storage of data on media such as magnetic disk, or to a description of data that depends on such physical factors as length of elements and records, pointers, etc.

PRIVACY

The prevention of unauthorized usage or dissemination of personal data.

QUERY LANGUAGE

A language, usually English-like, that enables an end user to interact directly with a database management system, and to retrieve and possibly modify data stored in a database.

RECORD

See [ANSI 77, NBS 77].

RECORD TYPE

The category to which a record belongs by virtue of a record format defined in the database schema. All records of a given type have the same format.

RECOVERY

The restoration of all or part of a database to a consistent state after an error or failure has occurred.

RELATIONAL MODEL

A data model providing for the expression of relationships among data elements as formal, mathematical relations. Informally, a relation appears as a table of data representing all occurrences of the relationship among the data elements or attributes of the relation. A row of the table, called a tuple, comprises one occurrence analogous to a record in conventional terminology.

REORGANIZATION

A major change in the way a database is logically and/or physically structured.

RETRIEVAL

The process of obtaining stored data from a database. The process includes the operations of identifying, locating, and transferring the data.

SCHEMA

A complete description of a database, written in a data definition language, that is processed and stored by a DBMS.

SECURITY

The protections provided to prevent unauthorized or accidental access to a database or its elements, or the updating, copying, removal, or destruction of the database or the database management system.

SERIAL FILE

A file where records are ordered in sequence (sorted) according to the values of one or more key fields in each record, and where the records are physically stored adjacent to one another according to this sequence.

SEQUENTIAL FILE

A file where records are ordered according to the values of one or more key fields, and the records are processed in this sequence from the beginning of the file. The records are not necessarily adjacent to one another, e.g., if stored on magnetic disk.

SET (CODASYL)

In a network or CODASYL type of database, a named logical relationship between record types, consisting of one owner record type, one or more member record types, and a prescribed order among the member records.

SUBSCHEMA

A description of selected data elements and particular relationships among them, as used by one or more application programs.

SUBSCHEMA DATA DEFINITION LANGUAGE

A computer language that is used to define a subschema, using as a basis an existing schema.

TRANSACTION

A command, message, or input record which explicitly or implicitly calls for a processing action, such as updating a file.

REFERENCES

[ANSI 77], American National Standards Committee on Computers and Information Processing (X3), *American National Dictionary for Information Processing*, X3/TR-1-77, Computer and Business Equipment Manufacturers Association, Washington, DC, 1977, see also [NBS 77].

[BAKE 75], Baker, F. T., "Structured Programming in a Production Programming Environment," *Proceedings of the International Conference on Reliable Software*, (available from IEEE Computer Society, Long Beach, CA), April 21-23, 1975, 172-185.

[BLEI 67], Bleier, R. E., "Treating Hierarchical Data Structures in the SDC Time-Shared Data Management System (TDMS)," *Proceedings of the ACM National Conference*, Association for Computing Machinery, NY, 1967, 41-49.

[BOEH 78], Boehm, B. et al., *Characteristics of Software Quality*, North-Holland Publishing Company, NY, 1978.

[BRAN 80], Branstad, M. A., Cherniavsky, J. C., and Adrion, W. R., *Validation, Verification, and Testing for the Individual Programmer*, NBS Special Publication 500-56, February 1980.

[BROO 75], Brooks, F. P., Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, MA, 1975.

[BUCK 79], Buckley, Fletcher, "A Standard for Software Quality Assurance Plans," *Computer*, 12, 8 (August 1979), 43-50.

[CODA 76], CODASYL Systems Committee, *Selection and Acquisition of Data Base Management Systems*, (available from ACM, NY), 1976.

[CODA 78], CODASYL Data Description Language Committee, *Journal of Development*, (available from Materiel Data Management Branch, Dept. of Supply and Services, Canadian Govt., Hull, Que., Canada), Jan. 1978.

[CODD 70], Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 13, 6 (June, 1970), 377-397.

[COMP 76], The Comptroller General, General Accounting Office, *Lessons Learned About Acquiring Financial Management and Other Information Systems*, August, 1976.

[COMP 77a], The Comptroller General, General Accounting Office, *Problems Found With Government Acquisition and Use of Computers From November 1965 to December 1976*, March 15, 1977.

[COMP 77b], The Comptroller General, General Accounting Office, *Millions in Savings Possible in Converting Programs From One Computer to Another*, September 15, 1977.

[COMP 79a], The Comptroller General, General Accounting Office, *Data Base Management Systems—Without Careful Planning There Can be Problems*, June 29, 1979.

[COMP 79b], The Comptroller General, General Accounting Office, *Contracting For Computer Software Development—Serious Problems Require Management Attention to Avoid Wasting Additional Millions*, November 9, 1979.

[COMP 80a], The Comptroller General, General Accounting Office, *Wider Use of Better Computer Software Technology Can Improve Management Control and Reduce Costs*, April 29, 1980.

[COMP 80b], The Comptroller General, General Accounting Office, *Conversion: A Costly Disruptive Process That Must Be Considered When Buying Computers*, June 3, 1980.

[DATE 77], Date, C. J., *An Introduction to Database Systems*, (second edition), Addison-Wesley, Reading, MA, 1977.

[EEC 80a], European Economic Community, *Database Administration: Experience from a European Survey*, (available from Commission of the European Communities, Brussels, Belgium), 1980.

[EEC 80b], European Economic Community, *Maintenance of Applications*, (available from Commission of European Communities, Brussels, Belgium), 1980.

[EEC 80c], European Economic Community, *Selection and Evaluation of Data Base Management Systems*, (available from Commission of European Communities, Brussels, Belgium), 1980.

[FAGA 76], Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, 15, 3, 1976, 182-211.

[FIFE 77], Fife, D. W., *Computer Software Management: A Primer for Project Management and Quality Control*, NBS Special Publication 500-11, July 1977.

[FRY 76], Fry, J. P., and Sibley, E. H., "Evolution of Data-Base Management Systems," *Computing Surveys*, 8, 1 (March 1976), 7-42.

[GANE 79], Gane, C., and Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

[HARD 80], Hardgrave, W. T., "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Philosophies," *IEEE Transactions on Software Engineering*, SE-6, 4 (July 1980), 357-372.

[IBM 74a], IBM Corporation, Data Processing Division, *Improved Programming Technologies—An Overview*, Installation Management Report GC20-1850-0, October 1974.

[IBM 74b], IBM Corporation, Data Processing Division, *HIPO—A Design Aid and Documentation Technique* Installation Management Report GC20-1851-0, October 1974.

[ICST 77], Institute for Computer Sciences and Technology, National Bureau of Standards, *A Survey of Eleven Government Developed Data Element Dictionary/Directory Systems*, NBS Special Publication 500-16, August 1977.

[ICST 80], National Bureau of Standards, *Prospectus for Data Dictionary System Standard*, NBSIR 80-2115, September 1980.

[INFO 79], *Infosystems Magazine*, "User Survey: DBMS Is Still a New Technology," 26, 9 (September 1979), 70-74.

[KERN 74], Kernighan, B. W., and Plauger, P. J., *The Elements of Programming Style*, McGraw-Hill, NY, 1974.

[LEFK 77], Lefkovits, Henry C., *Data Dictionary Systems*, Q. E. D. Information Sciences, Inc., Wellesley, MA, 1977.

[LEON 77], Leong-Hong, B., and Marron, B., *Technical Profile of Seven Data Element Dictionary/Directory Systems*, NBS Special Publication 500-3, February 1977.

[LEON 78], Leong-Hong, B., and Marron, B., *Database Administration: Concepts, Tools, Experiences, and Problems*, NBS Special Publication 500-28, March 1978.

[LYON 80], Lyon, G. (Ed.), *Using ANS FORTRAN*, NBS Handbook 131, March 1980.

[MART 76], Martin, J., *Principles of Data-Base Management*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[MART 77], Martin, J., *Computer Data-Base Organization, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[MCGE 77], McGee, W. C., "The Information Management System IMS/VS," *IBM Systems Journal*, 16, 2, 1977, 84-168.

[MICH 76], Michaels, A. S., Mittman, B., and Carlson, C. R., "A Comparison of Relational and CODASYL Approaches to Data-Base Management," *Computing Surveys*, 8, 1 (March 1976), 125-151.

[NBS 76], National Bureau of Standards, *Guidelines for Documentation of Computer Programs and Automated Data Systems*, Federal Information Processing Standards Publication 38, February 15, 1976.

[NBS 77], National Bureau of Standards, *Dictionary for Information Processing*, Federal Information Processing Standards Publication 11-1, September 30, 1977.

[NBS 79], National Bureau of Standards, *Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase*, Federal Information Processing Standards Publication 64, August 1, 1979.

[NBS 80a], National Bureau of Standards, *Guidelines for Security of Computer Applications*, Federal Information Processing Standards Publication 73, June 30, 1980.

[NBS 80b], National Bureau of Standards, *Guideline for Planning and Using a Data Dictionary System*, Federal Information Processing Standards Publication 76, August 20, 1980.

[NEUM 77], Neumann, A. J., *Features of Seven Audit Software Packages—Principles and Capabilities*, NBS Special Publication 500-13, July 1977.

[OLLE 78], Olle, T. W., *The CODASYL Approach to Data Base Management*, John Wiley and Sons, Chichester, England, 1978.

[ROTH 77], Rothnie, J. B., and Goodman, N., *An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases*, Technical Report CCA-77-04, Computer Corporation of America, May 1977.

[SOCK 79], Sockut, G. H., and Goldberg, R. P., *Data Base Reorganization—Principles and Practice*, NBS Special Publication 500-47, April 1979.

[TAYL 76], Taylor, R. W., and Frank, R. L., "CODASYL Data-Base Management Systems," *Computing Surveys*, 8, 1 (March 1976), 67-104.

[TSIC 76a], Tsichritzis, D. C., and Lochovsky, F. H., "Hierarchical Data-Base Management," *Computing Surveys*, 8, 1 (March 1976), 105-124.

[TSIC 76b], Tsichritzis, D. C., and Lochovsky, F. H., *Data Base Management Systems*, Academic Press, New York, 1976.

[TSIC 78], Tsichritzis, D. C., and Lochovsky, F. H., "Designing the Data Base," *DATAMATION*, August 1978.

[ULLM 80], Ullman, J. T., *Principles of Database Systems*, Computer Science Press, Inc., Potomac, MD, 1980.

[VanT 74], Van Tassel, D., *Program Style, Design, Efficiency, Debugging, and Testing*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[WALS 77], Walston, C. E., and Felix, C. P., "A Method of Program Estimation and Measurement," *IBM Systems Journal*, 16, 1, 1977.

[YOUR 75a], Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[YOUR 75b], Yourdon, E., "A Case Study in Structured Programming: Redesign of a Payroll System," *Digest of Papers of IEEE COMPCON*, (available from IEEE Computer Society, Long Beach, CA), September 9-11, 1975.

U.S. GOVERNMENT PRINTING OFFICE : 1981 O - 331-191

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $13; foreign $16.25. Single copy, $3 domestic; $3.75 foreign.

NOTE: The Journal was formerly published in two sections: Section A "Physics and Chemistry" and Section B "Mathematical Sciences."

DIMENSIONS/NBS—This monthly magazine is published to inform scientists, engineers, business and industry leaders, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing. Annual subscription: domestic $11; foreign $13.75.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Services, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Services, Springfield, VA 22161, in paper copy or microfiche form.

U.S. DEPARTMENT OF COMMERCE
National Technical Information Service
5285 Port Royal Road
Springfield, Virginia 22161

OFFICIAL BUSINESS

POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-211

3rd Class Bulk Rate