1999

Management of Software Engineering, The - Part I: Principles of Software Engineering

Harlan D. Mills

Reprinted from

IBM Systems Journal

Volume Ninteen | Number Four | 1980

The management of software engineering
Part I: Principles of software engineering

by H. D. Mills

Software engineering may be defined as the systematic design and development of software products and the management of the software process. Software engineering has as one of its primary objectives the production of programs that meet specifications, and are demonstrably accurate, produced on time, and within budget. This paper in five parts discusses the principles and practices used by the IBM Federal Systems Division for the design, development, and management of software.

The general principles of software engineering are set forth in Part I, in which the author relates software engineering to the whole field of the system development process—system engineering, hardware engineering, software engineering, and system integration. Presented briefly are overviews of the major aspects of software engineering—design, development, and management.

Part II, on the software engineering program, deals with the architecture of the new discipline. Discussed is the underlying concept of the software development life cycle. Based upon this foundation are a series of formally documented practices that set forth the specifics of software design, development, and management methods, which are presented in this paper. Also presented is an educational program whereby this discipline with its principles and practices has been made teachable.

Part III, on software engineering design practices, deals with activities bounded by requirements definition on one side and program implementation on the other. Three levels of design practices are defined, dealing with construction and verification of software systems, modules within systems, and individual programs. At each stage, a new level of mathematical rigor and precision for creating and evaluating software designs is introduced.

Part IV, on software engineering development practices, discusses a methodology for translating designs into software products. The subject is treated under two main headings, code management and integration engineering. These are rigorous methods for building the parts and integrating them into the whole software product that meets the design specifications.

Part V deals with the management of software engineering, which is primarily the intellectual control of the whole software engineering process. Intellectual control is brought about by a technical review strategy, a cost management approach, and a project environment for effective software development.

The management of software engineering Part I: Principles of software engineering

by H. D. Mills

In the past 20 years, the Federal Systems Division of the IBM Corporation has been involved with some of the nation's most complex and demanding software developments. These include the ground support software for the NASA Manned Space Series of the Mercury, Gemini, Apollo, and Skylab Programs (reaching the moon with Apollo), and both the ground and space software for the NASA Space Shuttle Program. FSD has also developed software for the Safeguard Anti-Ballistic Missile System, for the Enroute Traffic Control System for the FAA, and many other major civil and defense systems.

Software engineering began to emerge in FSD some ten years ago in a continuing evolution that is still underway. Ten years ago general management expected the worst from software projects—cost overruns, late deliveries, unreliable and incomplete software. Today, management has learned to expect on-time, within-budget deliveries of high-quality software. A Navy helicopter/ship system, called LAMPS, provides a recent example. LAMPS software was a four-year project of over 200 person-years of effort, developing over three million and integrating over seven million words of program and data for eight different processors distributed between a helicopter and a ship, in 45 incremental deliveries. Every one of those deliveries was on time and under budget. A more extended example can be found in the NASA space program, where in the past ten years, FSD has managed some 7000 person-years of software development, developing and integrating over a hundred million bytes of program and data for ground and space processors in over a dozen projects. There were few late or overrun deliveries in that decade, and none at all in the past four years.

There have been two evolutions in FSD: first, an evolution in ideas, leading to a growing discipline in both the management and technical sides of software engineering, and second, an evolution in the number and skill of people using the discipline. This evolution has not been without pain and attrition. Software is a new subject of human endeavor. Just as programming has evolved from a cut and try individual activity to a precision design process in structured programming, software engineering has evolved from an undependable group activity to an orderly and manageable activity for meeting schedules and budgets with high-quality products.

It is one thing to talk about orderly software development, and quite another to achieve it. The basis for this orderly control is mathematical discipline, even though the problem being solved by the software may not be mathematical. The key management standards of software engineering in FSD are based on mathematical theorems about how programs can be structured, documented, and organized into larger systems, because without theorems for bedrock, choices reduce to matters of management style and individual experience.

The FSD Software Engineering Education which supports the Program is highly mathematical for both managers and programmers. Set theory, logic, mathematical functions, and state machines play key roles in education, not for the sake of mathematics itself, but because practical experience has shown that that level of precision is required in order to do more than talk about orderly software development.

The present state of the FSD Software Engineering Program is described in the accompanying papers.

"Software engineering program," by D. O'Neill
"Software engineering design practices," by R. C. Linger
"Software engineering development practices," by M. Dyer
"Software engineering management practices," by R. E. Quinnan

What is software?

Software began as a synonym for computer programs, but the term has taken on a much more extensive meaning. The effective use of computer hardware requires more than programs. It requires well-informed users and human procedures for computer operations, data entry, and program execution. These requirements call for instructions for humans of no less precision and completeness than programs for the computers. Thus, operators' guides, users' guides, etc. become as important to a system operation as programs. Further, the users must understand well enough what the computers do to correctly interpret their outputs and intelligently prepare their inputs to meet operational objectives. Thus, requirements and specifications of computer programs and systems are of vital importance to the users as well.

Although computers began as single units serving a single user at a time, the rapid growth of multi/distributed processing systems to serve multi/distributed users has greatly expanded the role of software. Software is the logical glue that can hold many computers and digital devices of all kinds together in a coherent system, which in turn interacts with many kinds of people—clerical, professional, staff specialists, and management—in the operation of an enterprise.

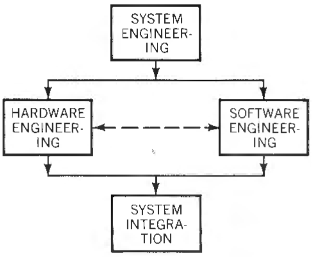
As a result of the pervasive role of software in a multi/distributed processing system, it seems proper to redefine the term software from its usual meaning of single programs to mean logical doctrine for the harmonious cooperation of a system of people and machines—usually many kinds of people and many kinds of machines. In such a system, the agents of action are people and machines, with the blueprints for their action supplied by software. A human procedure is as important to the system as a machine procedure. People have radically different instruction sets than machines, including an operation called "use your common sense," but they have instruction sets just the same. The synchronization of two people or a person and a machine is as important as the synchronization of two machines, but people often supply self-synchronization capabilities. Even "off the shelf machines" have an analog in "people with presently available skills."

Thus, software consists of operational requirements for a system, its specifications, design, and programs, all its user manuals and guides, and its maintenance documentation. Further, this whole software complex needs to evolve as a consistent whole as the operation evolves, as new hardware is added, and as new people are added. That is, software is typically a set of logical blueprints for the operation and use of a multi/distributed processing system by an organization of people in its natural evolution over time.

What is software engineering?

Software engineering is a growing set of disciplines and procedures for the dependable development and maintenance of software, as embodied in the FSD Software Engineering Practices, and discussed in Reference 1. For a wider perspective, we can identify the following four definite functions in an overall system development process, the relationships among which are illustrated in Figure 1.

Figure 1 System development



Software engineering stands between system engineering and system integration, accepting from system engineering the system software requirements and resources, and providing system integration with the software for meeting those requirements with those resources. Thus the total software of a system is a joint product of system engineering and software engineering, which begins with a defined system purpose and a defined configuration of hardware.

Of course, operating systems, compilers, and programming support systems all represent special and specialized software system developments, and the disciplines and procedures of software engineering apply fully to them. But we are usually more preoccupied with application systems, which make use of such support systems as extensions of the hardware.

The FSD practices classify the disciplines of Software Engineering into the following three categories:

- *Design*—system design, module design, program design, and data design, all of which culminate in source code in one or more compilable programming languages, as well as in linkage editor, loader, and job control languages.
- *Development*—organization of design activities into sustained software development, selection, and control of design support facilities, code management, test, and software integration planning and control.
- *Management*—work breakdown and organization procedures, estimation, and scheduling of personnel and computer resources required for software design and development, measurement and control of software design and development.

Software engineering design

Attention to the principles of software design has focused on three distinct areas during the past decade and has resulted in an abundance of useful and well-tested material on the following subjects:

- Sequential process control—characterized by structured programming and program correctness ideas of Dahl, Dijkstra, and Hoare,[2] Hoare,[3] Linger, Mills, and Witt,[4] and Wirth.[5,6]
- System and data structuring—characterized by modular decomposition ideas of Dahl, Dijkstra, and Hoare,[2] Ferrentino and Mills,[7,8] and Parnas.[9].
- Real-time and multiple/distributed processing control—characterized by concurrent processing and process synchronization ideas of Brinch Hansen,[10] Hoare,[11] and Wirth.[12]

Software design requires the integration of these three areas into a systematic process, as discussed in Reference 13. These design principles provide increased discipline and repeatability for the design process. Designers can understand, evaluate, and criticize each other's work in a common, objective framework. As pointed out by Weinberg,[14] people can better practice egoless software design by focusing criticisms on the design and not on the author. These design principles also establish the criteria for more formalized design inspection procedures that permit designers, inspectors, and management to better prepare, conduct, and interpret the results of periodic design inspections.

Software engineering development

Although the primary thrust of software engineering is embodied in design, the organization and support of design activities into sustained software development is an equally important activity, as discussed in References 1, 15, and 16. The selection of design and programming languages and their support tools, the use of library support systems to maintain and monitor a design under development, and the implementation of a test and integration strategy will all affect the design process in major ways. The disciplines and procedures needed to sustain software development must be scrutinized and chosen as carefully as design principles.

Intellectual control is the key to orderly software development. It is made possible by a sequence of logically equivalent software descriptions, beginning with high-level specifications and proceeding through successively lower-level specification refinements until the level of source code is reached. Successive descriptions can be baselined and validated to milestones, so that the intermediate progress of software development is more visible to management. This activity of creating a sequence of more and more detailed specification refinements of an initial specification is the process of *top-down development*.

The intellectual control and management of design abstractions and details is the basis for the development discipline. Design and programming languages are required that can deal with procedure abstractions and data abstractions, with system structure, and with the harmonious cooperation of multi/distributed processes. Library support systems are required that can handle the convenient creation, storage, retrieval, and correction of design units, and provide the overall assessment of design status and progress against objectives.

The first guarantee of quality in design is in well-informed, well-educated, and well-motivated designers. Quality must be built into designs, and cannot be inspected in or tested in. Nevertheless, any prudent development process verifies quality through inspection and testing. Inspection by peers in design, by users or surrogates, by other financial specialists concerned with cost, reliability, or maintainability not only increases confidence in the design at hand, but also provides designers with valuable lessons and insights to be applied to future designs. The very fact that designs face inspections motivates even the most conscientious designers to greater care, deeper simplicities, and more precision in their work.

Software engineering management

Management from a software engineering viewpoint is primarily the management of a design process, and represents an equally difficult intellectual activity. While the process is highly creative, it must still be estimated and scheduled, so that the various parts of the design activity can be coordinated and integrated into a harmonious result, and so that users and other functions of system development can plan on this result. The intellectual control that comes from well-conceived design and development disciplines and procedures is invaluable in this process. Without that intellectual control, even the best managers face hopeless odds in trying to see the work through.

To meet cost/schedule commitments based on imperfect estimation techniques, a software engineering manager must adopt a manage-and-design-to-cost/schedule process. That process requires a continuous and relentless rectification of design objectives with the cost/schedule needed to achieve those objectives. Occasionally, a brilliant new approach or technique which increases productivity and shortens time in the development process may simplify this. But usually, the best possible approaches and techniques have already been planned, and a shortfall or windfall in achievable software requires consultation with the user to make the best choices among function, performance, cost, and schedule. The intellectual control of software design not only allows better choices in a current development, but also stimulates subsequent improvements in function or performance for a well-designed baseline system resulting from the current development.

In software engineering, there are two parts to an estimate—making a good estimate and making the estimate good. The software engineering manager must see that both parts are right in addition to ensuring the right function and performance. That is not an easy task and never will be, but there is help on the way, as described in the companion articles and in the references.

ACKNOWLEDGMENTS
The authors thank FSD President John B. Jackson for giving them as well as other developers and students of the software engineering program the leadership and means to implement this program. We also thank James A. Bitonti for setting for us the goal of developing a written base of procedures for the educational program and project compliance accountability.

The author is located at the IBM Federal Systems Division, 10215 Fernwood Road, Bethesda, MD 20034.

The management of software engineering Part II: Software engineering program

by D. O'Neill

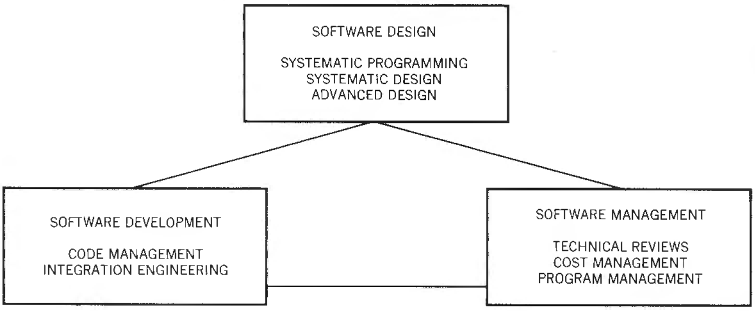
The breadth of applications in industry today stresses software development and has resulted in a diversity of design technologies, computer products, programming languages, support software tools, and documentation requirements. Moving from one application area to another can require major adjustments by both technical people and management. These time- and energy-consuming adjustments introduce more diversity and further complicate an already complex process.

It is well known that software costs associated with computer system developments have been increasing and are becoming a critical cost element in these developments.[17,18] Budget data indicate that software costs may become ninety percent of system development costs by 1985. Yet the development of reliable software on schedule within cost has been and remains a significant management challenge.[19,20] At the same time, hardware manufacturing costs are being reduced by orders of magnitude. These trends are introducing new levels of complexity into software developments as demands for system performance and reliability are requiring greater precision in software design and development. Potential solutions to these problems have been surveyed by the IBM Federal Systems Division, with particular attention to recent developments in the academic and professional communities.[7,13,21-23]

The result of our research into these new developments has been to make these developments teachable and practical in the discipline of *software engineering*. Software engineering has been defined as the systematic design and development of software products and the management of the software process. The software engineering discipline combines design topics resulting from university influences with the software development and management expertise of industry. Both perspectives are necessary to support a software engineering program that blends technology advances with practical innovations.

The software engineering program of the IBM Federal Systems Division demonstrates a commitment to the improving of the software development process beyond the software technology innovations of structured programming, top-down development,

Figure 1 Software engineering practices organization



and chief programmer teams. Following the advances in hardware technology, this program is designed to teach the practice of higher levels of precision in software design and development.[1,4,24,25] The program addresses current trends in the business environment that demand software product quality and reduced software costs.

The software engineering program combines development of comprehensive practices, education in such practices, and development of necessary support tools. The primary thrust of the program is the preparation of uniform *software engineering practices* that apply modern design, practical development, and proven management methods. Software engineering practices are introduced through formal education to provide a broad base of professional programmers who are able to produce software systems utilizing these disciplines. Uniform tools are provided to support the uniform practices. Process assessment has also been introduced to ensure that this program is being followed and to measure its effectiveness.

A comprehensive collection of technical and managerial practices is emerging from the combination of successful experience and university research. These represent the best of the current understanding of the software engineering process and a proven way of designing, developing, and managing software. Each practice defines specific work products that serve as visible intermediate steps in the process. The application of the design techniques produces modular designs and structured programs that are reliable and efficient as well as adaptable to change. Software development utilizes high-level languages and programming support library hierarchies to manage code produced in a natural sequence of phased increments. Management methodology provides plans and controls that ensure cost and schedule visibility of the process, as well as technical performance measurements of the emerging product. The relationships among software engineering design, development, and management practices are shown in Figure 1 and are discussed later in Parts III to V.

The software design practices introduce advanced software technology including systematic programming, systematic design, and advanced design. Systematic programming practices involve logical and program expression for recording designs, program design through stepwise refinement, and program verification using formal proof of correctness, as well as less formal methods. Systematic design practices cover data design with data types and structures, and modular design using state machines and modules. Advanced design practices cover concurrent design including synchronization and real-time considerations, and software system specification using state machine methodology.

Software development practices include code management and integration engineering. Code management ensures that software is uniform with respect to programming language usage, coding standards, and conventions. This also includes software system-building procedures and covers computer product support software and the software development environment. Integration engineering introduces procedures for software integration, incremental software development, and interface specification management. It also covers simulation and performance measurement software.

Software management practices include technical reviews, cost management, and program management. The technical reviews are product based for the completion of each work component of the software development life cycle. Cost management includes the practices for process and design-to-cost methodology, and program management is designed to improve the visibility of the software process through more effective plans and controls.

Software development life cycle

A set of *activities* has been defined that describes the software process from system definition through operational support. The activities that make up the software process are further defined in terms of work components that identify the tasks to be performed, as shown in Table 1. These activities portray software in the enlarged perspective of the full life cycle as viewed by the customer, and provide the basis for effective management. The activities may overlap in time, but each must be scheduled for completion prior to subsequent dependent activities. Many developments call for performance in all activities of the life cycle, whereas others may involve only certain ones. Work components have specific completion criteria in the form of *work products* that are subject to technical review.

Table 1 Software life-cycle activities

Activity	*Work components*
System definition	Software requirements definition Software system description Software development planning
Software design	Functional design Program design Test design Software tools Design evaluation
Software development	Module development Development testing
Software system test	Software system test procedures Software integration and test
System and acceptance test	System test support Acceptance test support
Operational support	System operation support Training Site deployment support

In many projects, support activities that are required to produce a work product may be collected together from an organizational or cost accounting viewpoint into a general support function. Typically, this includes project management, software configuration management, software quality assurance, software cost engineering, administrative centers, technical publication,[22] financial management services, and data management. Software engineering practices apply across the full life cycle. The correspondence between software engineering practices and the life-cycle activities is shown in Table 2.

System definition includes definition and analysis of the software system requirements, establishment of a software system description, and initiation of the software development planning necessary to proceed with further development of the software system. The software system requirements are a record of the complete system capability, including both the software and the environment in which the system is to operate. Because requirements documentation is expressed in natural language and may lack precision, a description that is produced for the software system only is prepared in a precise, detailed, succinct, and sufficient manner using prescribed methods of expression. The software system description must be traceable to the requirements document and must maintain semantic correspondence with that document. The initial software development plan is prepared on the basis of the software system description and includes cost management planning, schedules and external dependencies, and resources of both people and machines.

Table 2 Practices and activities relationships

Life cycle activities	*Design*			*Development*		*Management*		
	Advanced design	*Systematic design*	*Systematic programming*	*Code management*	*Integration engineering*	*Technical reviews*	*Cost management*	*Program management*
System definition	●					●	●	●
Software design		●	●	●	●	●	●	●
Software development			●	●	●	●	●	●
Software system test				●	●	●	●	●
System and acceptance test				●		●	●	●
Operational support				●			●	●

Software design includes conversion of the software system description into a design, design evaluation, preparation of test designs, and production of software tools. Functional designs are composed of module designs produced according to systematic design practices, and program designs are composed of structured programs produced from the module designs according to systematic programming practices. The preparation of test designs is performed using integration engineering practices.

The *software development* activity includes module development and development testing. Module development is the final elaboration of design details according to systematic programming practices and the preparation of source language statements that can be translated into executing code. These statements comply with their program and module designs and are produced in accordance with code management practices. Module testing includes test procedure executions to ensure that an implemented module complies with the specification of the software system and is conducted according to integration engineering practices.

Software system testing includes the preparation of testing procedures followed by software integration and testing as specified in the integration engineering practices. This is to ensure that the implemented software system complies with specification of the software system and the code management practices.

System and acceptance testing ensures that the software system complies with all project-deliverable objectives. This testing also verifies that all deliverable items exist and all reviews have been successfully completed. Code management practices apply during this activity.

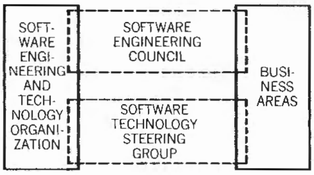
Operational support includes system operation support, site deployment support, and training; code management practices apply here as well. The product of a project is typically delivered to a customer who operates it with minimal post-delivery assistance. Customer procedures for change control, system evaluation, and so forth apply during the operations stage of the life cycle.

Software engineering program implementation

The main thrust of the software engineering program is to improve product quality and reduce cost by implementing consistent practices. A successful operation of the program also requires education, tools, and measurements. Education in modern techniques gives personnel an understanding and appreciation of the methods defined in the practices. Tools that support these methods ensure their effective utilization and rapid adoption. The application of these methods and realization of their benefits require continuous assessment and feedback of results.

The software engineering program requires continuous communication between the business areas responsible for contract performance and the Software Engineering and Advanced Technology group, which is the divisional organization responsible for technology advances. Two special communication channels, shown in Figure 2, have been established for this purpose.

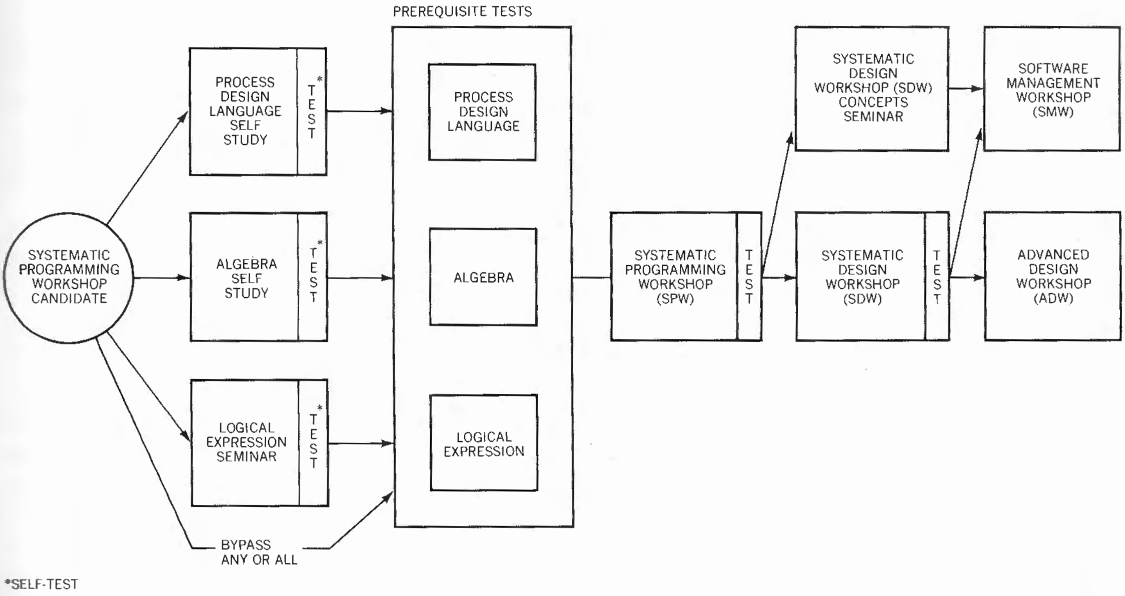
Figure 2 Software engineering program organization


SOFTWARE ENGINEERING AND TECHNOLOGY ORGANIZATION
SOFTWARE ENGINEERING COUNCIL
SOFTWARE TECHNOLOGY STEERING GROUP
BUSINESS AREAS

A Software Engineering Council provides policy guidance in software operations and is responsible for setting the direction of the software engineering program. The Council, which meets quarterly, is composed of senior software executives. A Software Technology Steering Group, which sponsors and reviews the software engineering practices, formulates software technology strategy. The latter group also generates the software investment programs. The steering group meets monthly and is composed of representatives from the business areas and the Software Engineering and Technology Group. They are responsible for formulating the software engineering practices to reflect a combination of the best current business usage and practical new ideas from the computer sciences. Participation of the business areas ensures their commitment to the program while at the same time providing the technologists with practical insights and project constraints.

A software engineering curriculum has been designed around seminars and workshops, as illustrated in Figure 3. Three prerequisite self-study courses and four instructor-taught courses are available in this curriculum, which was begun in October 1977 and is to be completed in 1981. Their common objective is to

Figure 3 Software engineering education curriculum


PREREQUISITE TESTS
SYSTEMATIC PROGRAMMING WORKSHOP CANDIDATE
PROCESS DESIGN LANGUAGE SELF STUDY
TEST*
ALGEBRA SELF STUDY
TEST*
LOGICAL EXPRESSION SEMINAR
TEST*
BYPASS ANY OR ALL
PROCESS DESIGN LANGUAGE
ALGEBRA
LOGICAL EXPRESSION
SYSTEMATIC PROGRAMMING WORKSHOP (SPW)
TEST
SYSTEMATIC DESIGN WORKSHOP (SDW) CONCEPTS SEMINAR
SYSTEMATIC DESIGN WORKSHOP (SDW)
TEST
SOFTWARE MANAGEMENT WORKSHOP (SMW)
ADVANCED DESIGN WORKSHOP (ADW)
*SELF-TEST

significantly improve the predictability of the software process and the quality of the resulting product. The underlying technical objective is to increase visibility and intellectual control over a developing software product by the process of stepwise refinement using standard conceptual models and a limited number of basic control and data structures. The process calls for the successive replacement of abstract designs with increasingly more detailed designs that are known to be equivalent. The underlying management objective is to provide complementary development strategies, feedback mechanisms, and control techniques. Admission to these courses is arranged through Software Engineering Program Coordinators in each business area. Table 3 summarizes information on the expected audience, prerequisites, and duration of each course.

The *three prerequisite self-studies* prepare students to read the professional literature and to communicate in the workshops using the language of mathematics and a software design language. Prerequisite tests are administered by business area coordinators in algebra, Process Design Language (PDL), and logical expression (basic concepts and the notations of set theory and symbolic logic).

A *Systematic Programming Workshop* (*SPW*) advocates a particular discipline for the design of sequential programs modeled on mathematical functions. Designs are expanded from abstract statements of a program's intended function, using PDL, which is a programming-like design language. At each step, a design state-

Table 3 Software education summary

Course	*Audience*	*Prerequisites*	*Duration*
Self-study	All programmers and analysts	Programming experience	Logical expression 10–15 hours Algebra 1/2–1 hour Process Design Language 6–10 hours
Systematic Programming Workshop (SPW)	All programmers and analysts and others designated by management	Algebra, Process Design Language Logical expression	8 1/2 days
Systematic Design Workshop (SDW)	Key programmers and analysts and others designated by management	Satisfactory completion of Systematic Programming Workshop	5 days
Advanced Design Workshop (ADW)	Software designers and architects designated by management	Satisfactory completion of Systematic Design Workshop	3 days
Software Management Workshop (SMW)	All software managers and selected technical personnel	Satisfactory completion of SDW or SDW concepts seminar	5 days

ment is replaced by a simple function-equivalent program whose components are simpler intended functions. By restricting these replacement programs to a limited set of program structures, the equivalence of successive versions of the design is more readily verified.

A *Systematic Design Workshop* (*SDW*) extends the stepwise-refinement discipline of SPW to include the design of sequential programs with retained data, modeled on finite-state machines. State machines provide for encapsulating collections of data, with access limited to a fixed set of related programs. Concepts are introduced that allow the designer to relate initial abstract representations of state data to later, more specific representations.

An *Advanced Design Workshop/Seminar* (*ADW/S*) extends the design concepts of SDW to include the design of concurrent systems, modeled on networks of communicating state machines. The designer is asked to view the overall system as a state machine, and then to partition state data to define a network of state machines. Each input is identified with a path through the network. Several options for introducing and controlling concurrency within this framework are discussed. The approach is applicable to a wide variety of hardware configurations.

The *Software Management Workshop* (SMW) includes a review of concepts of FSD's functional organization, and the specific role of the software engineering function. The primary emphasis is on the principles underlying FSD's software standards and practices, in the context of management of the software product, the technical methodology, the organization, the development environment, and the customer.

Education has been carried out by a small group of instructors drawn from the professional programming cadre in FSD. This was a departure from the traditional method of using full-time educators who have less actual programming experience. The courses themselves represented new offerings, much more technical than most internal training programs. The level of difficulty was also high, for job-related required courses. Nevertheless, student achievement has been excellent, and course evaluations by the students have been highly favorable.

A principal software investment priority is the development of the tools needed to reinforce software engineering and establish an environment of modern methods. Current emphasis is on two tools, a *system development laboratory* and a *programming support library*, because of their applicability to multiple facets of the program.

The software development laboratory establishes a more complete and uniform programming environment. The laboratory includes interactive, batch, and dedicated development facilities for design creation, program generation, simulation, and target machine execution. The implementation of this discipline uses proven off-the-shelf development software tools that are integrated to address the software development process. This approach isolates validation of system design to the software design activity, implementation to the software development activity, and software-hardware interaction to the system test activity. The laboratory approach provides the system developer with a single interface for the entire software development process.

The integrated development discipline uses commercially supported large-scale operating systems to aid in the reduction of life-cycle costs. With time-shared resources, small projects and large projects alike can use the total technology without incurring the expense of a large dedicated resource.

The programming support library helps organize and control a programming project. It serves as the means of communication among development personnel and forms a standard interface between programmers and the system development laboratory. The programming support library is designed to provide a complete hierarchical library facility. As such the library supports the code management and machinability needs associated with process design language during design, and it supports programming languages during module development, software integration, and product release. Thus this library is a key element in the application of software engineering technology. It is a collection of computer programs, disk-resident libraries, and operating procedures that provide facilities for programmers to store, edit, compile, and execute programs under development. Typically, the library produces summary statistics and analyses of such activities as storing, compiling, and executing programs.

Software engineering process assessment

One of the best ways of estimating future software development efforts and schedules is to rely on past experience. Yet few managers have access to recorded development data. A data base and retrieval system that can be used by line management to retrieve single-project development data as well as composite reports of selected categories of software makes possible a comparison of one manager's experience with that of managers of comparable efforts. The data base is also a source of information on software development.[3,4] In this role, it can support evaluations of the effectiveness of the software engineering program.

Individual projects may involve several categories of software, such as application, diagnostic, and support software. Application software can be further broken down by such characteristics as real-time signal processing, process control, and on-line graphics.

Data are associated with specific life-cycle activities, each of which is evaluated to determine key data parameters to be collected. Quantitative data and, more subjectively, project success judgments are included. The criteria for determining the pertinence of data are useful to software managers for characterizing and estimating the size of software projects.

Concluding remarks

Recent advances in software technology have necessitated a coordinated program involving people, tools, and practices. Education through a rich curriculum of software engineering courses is well underway. The development of tools that establish the proper environment for good programming is in progress, and software engineering practices are assembled and ready for use.

This program reflects an understanding of the software development process. With a realistic assessment of traditional methods, software engineering includes a comprehensive collection of technical and management practices that can be applied today. Beginning with software system requirements recorded in source libraries, advanced techniques are now available to permit the conceptual abstraction, modularization, and structuring of designs that reduce complexity to manageable proportions and improve the completeness and correctness of the resulting software product. Modern development tools and uniform code management practices support orderly code generation in high-level languages, with storage in hierarchical libraries from which patch-free, quality source code products are delivered. Integration engineering is emerging as a distinct organizational function with a foundation of advanced technology. Management practices provide the methodology for balancing cost, schedule, performance, and quality perspectives.

As a result, software product quality can be dramatically improved by the routine application of modern design practices that contribute directly to program correctness and error avoidance through simplified and understandable designs. The manageability of software can be improved through uniform practices that govern plans, controls, and cost management, and through technology innovations that greatly improve the visibility of the software product for more effective management and control. Productivity improvements result primarily from the improvements in software product quality and the elimination or reduction of software errors. This reduces error detection and correction during testing. In addition to simplified designs, broader usage of higher-level programming languages and improved support tools also contribute to productivity gains.

Taken together, the software engineering discipline is a broad attack on the problems faced by the software community.[26] By emphasizing methodology and theoretical foundations, FSD has attempted to establish a common level on which each individual can build to the full extent of one's own creative ability. The benefit has been steady improvement in measured results over a sustained period of time.

The author is located at the IBM Federal Systems Division, 18100 Frederick Drive, Gaithersburg, MD 20780.

The management of software engineering Part III: Software design practices

by R. C. Linger

It is well known that large-scale software development is a difficult and complex process that demands the best design and management techniques available. Without effective principles for structuring and organizing software design and development, even the best-managed projects can be overwhelmed by the sheer volume of logical complexity. What is not so well known is that with increasing frequency, large-scale software systems are being developed in an orderly and systematic manner according to new design and development principles, and that these systems are exhibiting remarkable quality in testing and use. The level of precision and rigor in their construction is itself remarkable, compared to what was thought possible just a few years ago.

A major forcing factor in this emerging human capability for logical precision on a large scale has been a dramatic increase, over the past decade, both in the availability of documented and tested principles for software design, and in the number of software professionals who understand and can apply them. These principles include the structured programming and program correctness ideas of Dahl, Dijkstra, and Hoare,[2] Hoare,[3] Linger, Mills, and Witt,[4] and Wirth;[5,6] the module and data design ideas of Dahl, Dijkstra, and Hoare,[2] Ferrentino and Mills,[7] and Parnas;[9] and the concurrent processing and synchronization ideas of Brinch Hansen,[10] Hoare,[11] and Dijkstra.[27] Effective management principles for organizing and controlling software development have emerged as well, as described in Mills[11] and Baker.[15]

In 1977, the Federal Systems Division of the IBM Corporation established a Software Engineering Program to create a set of uniform software practices dealing with software design, development, and management principles (as indicated in Part II, Figure 1), and to develop an educational curriculum based on the practices. The resulting practices (some thirty in all) are the product of an extensive review process and reflect the best thinking and judgement of experienced software practitioners brought together from across the division.

Each practice is a terse statement of a particular aspect of software technology, defined in terms of scope, objectives, area of application, methodology, and required work products. Each practice establishes a foundation for acceptable professional performance, but makes no attempt to educate to that foundation. That is the purpose of the educational curriculum. In fact, a practice may seem mysterious indeed, without a corresponding course to back it up!

In particular, the software design practices define principles for specification, design, and verification of software systems, modules, data, and programs. The principles provide means to maintain intellectual control over complex software developments. They have deep roots in mathematics, yet correspond closely to concepts that have long been part of effective software design. Their value lies in the increased discipline and order they bring to the design process, as well as in the improved quality of the resulting software products. The practices provide uniform expressive forms at each stage of design for better communication among software designers, managers, and users. They also provide objective criteria for design analysis and evaluation, as part of a continuing process of inspection and review.

The software design practices are organized into three groups, as shown in Table 1. Practices in the first group, *systematic programming*, deal with forms for recording individual program designs, as well as techniques for program construction and verification. Methods for organizing the synchronous logic of a software system into a hierarchy of *design modules* (special combinations of programs and permanent data) are defined in the second group, *systematic design*. Finally, the *advanced design* group prescribes techniques for overall software system specification and for the design of concurrent programs that must share resources and cooperate in execution.

Software design disciplines

Three of the practices, one from each group, form a logical progression of design disciplines, that is, program design, modular design, and real-time design. Each of these practices defines concepts for a particular level of expression within the overall software design process. The idea of these three practices, dealing with design of programs, modules, and concurrent systems, is not that of decomposition of subject matter. Rather the idea is that of a sequence of building block methodologies, each of which draws heavily on its predecessors.

Program design, the basic practice of the three, is concerned with programs that execute and transform data independently of data storage between executions. Modular design makes use of programs, with the one additional concept of the storage of data be-

Table 1 Software design practices

Systematic programming practices	*Purpose*
Logical expression	Prescribes mathematics-based techniques for precise expression and reasoning that apply to all phases of software development.
Program expression	Defines control, data, and program structures for recording program designs.
Program design	Specifies a process of stepwise refinement for recording structured program designs.
Program design verification	Prescribes function-theoretic techniques for proving the correctness of structured programs.
Systematic design practices	
Data design	Specifies the use of abstract data objects and operations in a high-level design framework.
Modular design	Defines techniques for designing synchronous software systems, based on state machines and design modules.
Advanced design practices	
Software system specification	Defines a process based on state machines for creating a specification as the cornerstone documentation of a software system.
Real-time design	Defines a stagewise process for designing asynchronous software to achieve correct concurrent operation, with optimization to meet real-time processing requirements.

tween executions. It permits the definition of a data processing service for a user (a person or another module), with data storage as an integral part of that service. A module is constructed out of program operations plus the designation of data to be retained (stored) after program executions. The real-time design practice makes use of modules with the one additional concept of the asynchronous control of concurrent module executions. That is, a system is constructed out of modules plus the designation of real-time priorities for their concurrent execution.

In consequence of this building block structure, the design activities of a software system are sharply defined at the three levels of program, module, and system. At the system level, one is concerned only with the control of modules, and defers matters of user specifications and services to the module level. At the module level, one defers matters of processing to the program level. In summary, these design practices have the following properties.

Program design is concerned with programs only, but with no permanent storage of data between program invocations. The logical model of a program is a *mathematical function*, which defines the input and output characteristics of the program, but not its internals. Elements of the program design practice are shown in Appendix A, as an example of the format and content of an FSD practice.

Modular design is concerned with a collection of program operations and persistent data storage facilities that (1) make up a complete service to some user, and (2) represent all permissible ways of affecting the persistent data. A module is incompletely defined if other programs can affect its persistent data in any way—other than through the services that the module provides. The logical model of a module is a *state machine* that defines (1) the collective input and output characteristics of all the program operations of the module and (2) the data that are persistent (i.e., the state of the state machine).

Real-time design is concerned with the coordination and synchronization of a collection of modules operating concurrently in a computing system, possibly with multiple processors, so that they (1) do not inadvertently interfere with one another, (2) meet real-time deadlines as required, and (3) make sufficiently efficient use of the computing system. The logical model of a concurrent system design is an *indeterministic state machine*, which reflects the various possible rates of execution of its constituent modules in providing acceptable system performance to the module users. In practice, a collection of modules may be initialized together, then run asynchronously for some period of time on demand from various users, and then quiesced together again. During this time, other modules may be initialized, run, and quiesced asynchronously.

The remaining software design practices support and extend this progression of design disciplines. The practices of the advanced design group are currently under development. The systematic programming and systematic design practices are now described in detail.

Systematic programming practices

logical expression

The logical expression practice specifies rigorous methods of reasoning and expression based on mathematical principles for use during system and program development. Logical expression includes the concepts, structures, operations, and notation of set theory, functions, and predicate logic. These expressive forms improve communication among designers and help clarify program requirements, specification, and design documentation.

They permit precision without vagueness in expressing design abstractions, while allowing the deferral of details to later phases of development.

It is possible to develop small programs without these standards and with less formality, but it is practically impossible to develop large software systems under sound engineering control at an acceptable level of reliability, productivity, and quality without an equivalent level of formality and logical precision. The logical expression practice is the conceptual foundation for other software engineering techniques, and helps develop familiarity with patterns of thought and notation found in software engineering literature.

Specifications in natural language often prove difficult to check for completeness, and design and implementation details can be easily and unintentionally mixed with the specification of processing requirements. But a compact mathematics-based notation (for example, sets and set operations) can help define precisely what is required at a uniform level of specification. The completeness of such a specification is more easily checked, and the eventual designs of specified objects (such as sequential or direct-access files) and operations (such as algorithms to test for set membership, add and delete members, etc.) are not influenced by premature detailing. Natural language explanations of the set operations can then be added for clarity, but with no requirement to carry the full burden of specification.

program expression

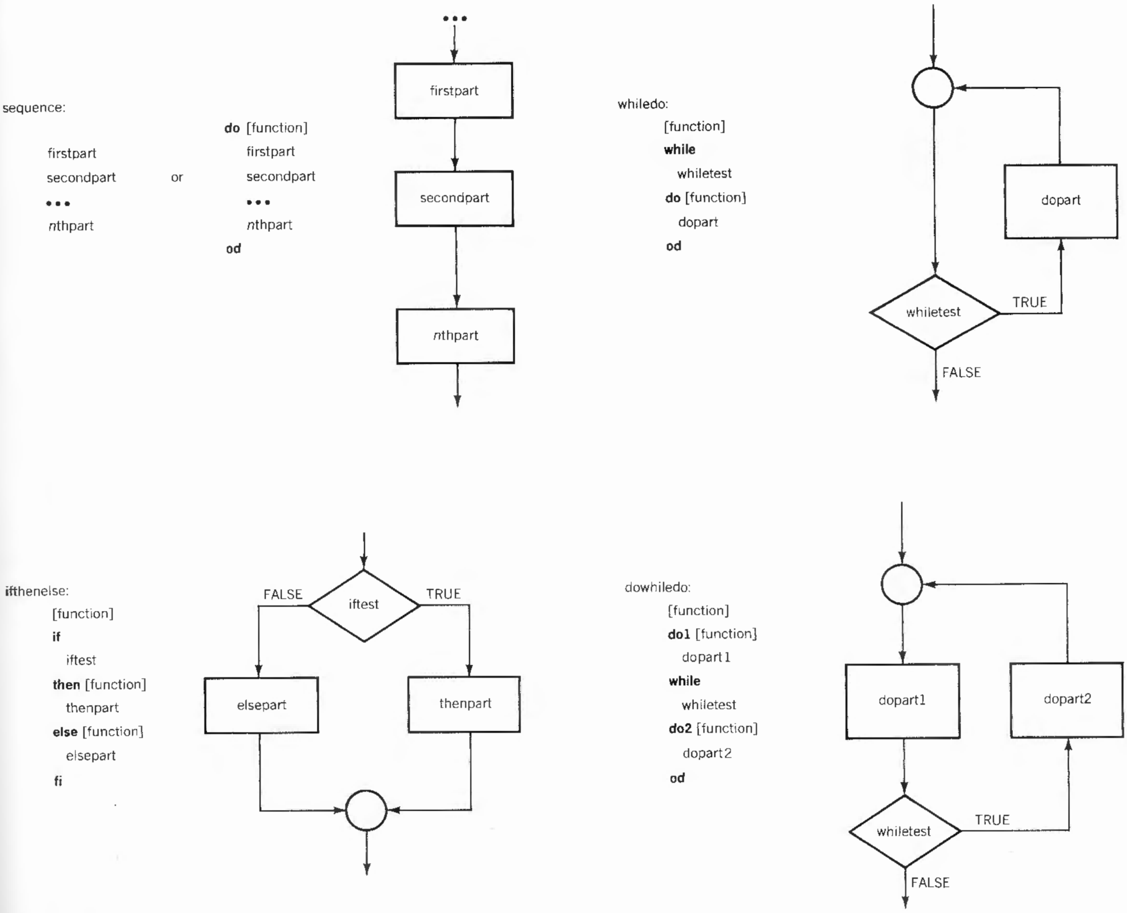
The program expression practice defines requirements for the textual language for recording program designs. This language is intended to support the following activities:

- Stepwise design of programs with correctness verification.
- Effective communication among users, designers, and developers.
- Reading, studying, and group reviewing of program designs.

The use of a design language helps to institutionalize the design process itself, so that design becomes a standard project activity, with its own intermediate work product between thought and code. For designers, there is time to schedule design progress, and for managers there is visible evidence of that progress. The problem of "*ad hoc* design in code" is superseded by a new medium and methodology for creating and reviewing the logical structures of a software system prior to implementation.

A definition of the Process Design Language (PDL) is maintained in an FSD bulletin, with a formal control board, as an example language that satisfies the requirements of the program expression practice. PDL is an open-ended specialization of natural

Figure 1 Some PDL control structures



```
sequence:

    firstpart                    do [function]
    secondpart         or            firstpart
    •••                              secondpart
    nthpart                          •••
                                     nthpart
                                 od
```

```
whiledo:
    [function]
    while
        whiletest
    do [function]
        dopart
    od
```

```
ifthenelse:
    [function]
    if
        iftest
    then [function]
        thenpart
    else [function]
        elsepart
    fi
```

```
dowhiledo:
    [function]
    do1 [function]
        dopart1
    while
        whiletest
    do2 [function]
        dopart2
    od
```

language, not a closed formal language,[4] and permits the designing of software from a logical point of view without getting into the physical storage and operations of specific computing systems. PDL permits precision for human expression and for direct human translation into target programming languages.

The principal specialization of PDL from natural language occurs in a standard *outer syntax* of control, data, and program structures, employing a few PDL keywords and a tabular typographic form. Outer syntax describes how operations are sequenced and controlled, how data are defined and accessed, and how programs are organized. A more flexible *inner syntax* of PDL deals with operations and tests. Outer syntax structures are applied with little or no variation from project to project, whereas inner syntax is intended to be specialized according to individual project needs.[28]

outer syntax

The outer syntax *control structures* of PDL include sequence, fordo, ifthen, ifthenelse, case, whiledo, dountil, and dowhiledo. Some of these structures are depicted in Figure 1 along with equivalent flowcharts; the structures are delimited by keywords shown in boldface, with parts indented for readability in larger contexts. In their effect on data, each of these single-entry/single-exit structures can be precisely described by a mathematical function that defines input and output characteristics, but describes nothing of internal operations. This function is known as the *program function* of the control structure.

In addition to control structures, PDL outer syntax provides high-level *data structures*, such as queues, stacks, sets, and sequences, together with conventions for their access.

Logical commentary, delimited in PDL programs by square brackets, is an important part of the design language. One type of logical commentary, known as *action* or *function commentary*, is used to record program functions, as shown in Figure 1. Function commentary can precede a control structure to define its function, or be attached to **do, then, else,** etc., keywords to define the function of the corresponding dopart, thenpart, elsepart, etc. Function commentary makes program designs self-documenting by recording intermediate abstractions in the design process. These abstractions make use of the expressive forms of the logical expression practice. The result is designs that can be read and understood at any level of detail.

PDL outer syntax *program structures* permit designs to be organized into hierarchies of small structured programs called *segments*. Each segment is delimited by keywords **proc** and **corp** and is of limited size (usually a page or less of text) and complexity. Segments are invoked in the hierarchy by statements of the following form:

run segmentname (parameter list)

Data objects are passed to and from segments in parameter lists, and local data, incidental to the function of a segment, are declared within the segment itself.

A miniature segment-structured program design is shown in Appendix B, along with logical commentary. Here function comments attached to **proc** keywords define the function of each segment. The operation **next** on the right of an assignment symbol (:=) reads a member from a sequence, and on the left writes a member to a sequence.

PDL programs are composed of individual control structures whose nesting and sequencing define a hierarchy in an *algebra of functions*. This function-theoretic algebra provides the principal source of power in structured programming, both by localizing and limiting the complexity of design decisions, and by providing a natural plan of attack for program reading, writing, and verification. In program reading, a control structure can be mentally replaced by its equivalent function with no side effects in other parts of the program. The containing control structure may then be likewise abstracted, and so on, to arrive at the function of the entire program. In program writing, functions can be expanded into equivalent control structures, again with no side effects elsewhere, continuing in this fashion until the entire program is elaborated in sufficient detail. Similarly, in verifying program correctness, a desired function and the actual function of the corresponding control structure can be compared for equivalence in a local setting, with no regard for program operations elsewhere. A PDL program is known to be correct when each of the control structures in its hierarchy has been shown to be correct.

inner syntax

The program expression practice imposes no restrictions on PDL inner syntax, beyond requirements for precision, conciseness, and understandability. Expressive forms for inner syntax must be chosen with the subject matter, level of design, and intended audience taken into account. For introductory design descriptions for general audiences, natural language may suffice. For precise communication of designs among professional programmers, more rigorous, mathematics-based forms may be required.

program design

The program design practice, depicted in Appendix A, specifies a function-based methodology for creating and recording correct program designs. As previously noted, the methodology is based on a view of structured programs as mathematical objects whose program functions form an algebra of functions. The starting point in the methodology is an *intended function*, which precisely defines the operations on data that a program is to carry out. It is a function definition in the mathematical sense, but may be described in English, mathematics, programming notation, or other expressive form. The principal operation in the practice is the replacement of an intended function by an equivalent structured program. Thus, an intended function of, for example,

z := maximum of x and y

appearing anywhere in an evolving program design, may be replaced by the following equivalent ifthenelse structure:

```
[z := maximum of x and y]
if
  x > y
then
  z := x
else
  z := y
fi
```

The ifthenelse carries out data transformations identical to the abstract intended function it replaces, which is carried forward into the expansion as a logical comment.

stepwise program refinement

In application, this process leads to *stepwise program refinement*,[5] in which a program design is developed as a hierarchy of control structure expansions, using the replacement of functions by equivalent expansions as the only rule of construction. A refinement step may consist of a single new control structure, or a miniature structured program composed of nested and sequenced control structures. Each refinement introduces new intended functions for subsequent refinement; resulting designs are hierarchical by construction. Data structures are also introduced in a hierarchical manner to support the local operations of each refinement. The program design *segment* is a natural unit of refinement for each step.

Stepwise refinement is not a mechanical process. A good understanding of overall program and data structures, from top to bottom, is required before recording segment designs. The best design is not the first design thought up, but the last; many iterations may be required to arrive at a suitable design structure. The depth of design varies with complexity. The design process is complete when further refinements become obvious.

A designer verifies the correctness of each refinement step by demonstrating that the program function of the refinement is equivalent to its intended function. The program function defines the actual data transformations carried out by the refinement; for correctness, the program function must match the intended function. Thus, verification is a two-step process: (1) derive the program function, then (2) compare it to the intended function. The program function may be self-evident and correctness determined by direct inspection. If the program function is not self-evident, a simpler design should be considered. Otherwise, verification techniques with sufficient rigor to determine correctness must be applied.

The program function of every segment should be defined in a logical commentary function comment. Important intermediate program functions should be recorded as well, including those for program parts that have been informally or formally proved correct.

The design of a small structured program in three refinement steps is shown in Appendix C. Intermediate functions are carried forward into successive versions as logical commentary, to document the design amid its detailing. Note the use of design language *multiple assignments* of the form a,b := c,d with meaning "compute values c and d and assign them to a and b, respectively."

program design verification

The program design verification practice defines methods to substantiate the correctness of program designs. Verification also assists in designing programs whose correctness is self-evident and in detecting logical errors, if any, in both intended functions and their corresponding program designs. Proofs may be carried out at either a formal, recorded level, or at an informal, unrecorded level of mental analysis.

A program design or design part is proved to be correct by proving that all its control structures are correct. The Correctness Theorem[4] summarizes function-theoretic proof requirements for the control structures of PDL. A control structure is proved to be correct by proving that its intended function is equivalent to (or a subset of) its program function. This demonstration is an intrinsic part of the stepwise refinement process, so that program designs *are both refined and shown to be correct in steps of manageable size*. Formal proofs of correctness based on the Correctness Theorem utilize systematic derivations and logical analysis to determine program functions of control structures and to compare them to intended functions. Formal proofs are recorded using a special proof syntax. Recording is important because formal proofs often contain insights not found in the program designs that are useful for subsequent design review and modification.

a proof example

A miniature illustration of a formal proof for a PDL sequence program design is shown in Appendix D. Part A defines the intended function f of the sequence, read "assign the values of y and x to x and y, respectively," that is, exchange x and y.

Part B is a sequence program composed of three PDL assignment statements (S1, S2, and S3). The Correctness Theorem states that to be correct, the intended function f must be equivalent to (or a subset of) the program function of the sequence, say p. The program function of a sequence program is computed by function composition. In this case, three functions are involved (composition denoted by "∘" symbol) as follows:

$$p = S3 \circ S2 \circ S1$$

That is, compute S1 output data values from S1 input, then S2 output from S2 input (equivalent to S1 output), then S3 output from S3 input (equivalent to S2 output). The program function defines S3 output in terms of S1 input.

Part C is the proof itself. The program function of a sequence program is derived by means of a systematic *trace table* with a numbered row for each assignment, and a column for each data item assigned (in this case x and y). Each table entry is an equation that relates values before the assignment to values after the assignment. For example, the first row defines x_1 (the value of x after the first assignment) as $x_0 + y_0$ (values before the first assignment) and also defines $y_1 = y_0$, that is, y is unchanged by the assignment.

Once the trace table equations are filled in, it is a simple matter to derive the final values of x and y (after the third assignment), i.e., x_3 and y_3, in terms of the initial values (before the first assignment), i.e., x_0 and y_0. As an example, if we write

$$x_3 = x_2 - y_2$$

and substitute expressions as follows:

$$\begin{aligned} x_3 &= x_1 - (x_1 - y_1) \\ &= y_1 \\ &= y_0, \end{aligned}$$

the final derivations for x and y are

$$x_3 = y_0 \text{ and } y_3 = x_0.$$

Therefore, the program function p is x,y := y,x. This function is equivalent to the intended function, and the program is indeed correct. The proof has been recorded for later study and analysis. Proofs for alternation and iteration control structures can be more complex than the sequence example, but the logical procedures to be followed in each case are known.

Informal proofs are carried out by asking and answering *correctness questions* that verbalize the correctness conditions of the formal proofs for each control structure. Informality does not connote a reduction in rigor; the correctness conditions to be proved are identical, whether formal or informal techniques are applied.

Systematic design practices

data design The data design practice specifies methodology for designing abstract data objects and operations.[29] Data abstractions provide a high-level design framework, and help keep the design process manageable because the designer deals with fewer concepts at a time. Design in terms of abstractions also permits changes in data representations to be made with minimal effect on the abstractions themselves.

Data types provide a basis for expressing data structures and the operations and tests that are permissible for those structures. The concept of data types can be applied repeatedly by stepwise refinements that introduce and focus on only a few structural and operational ideas at a time. Thus, data types permit very-high-level data structures and operations to be expressed in a form that the designer can refine into successively lower-level structures and operations, finally reaching an implementable level.

A data type is defined as a set of data objects and a set of operations and tests among those objects. A *scalar data type* defines data objects with no usable internal structure or parts. A *structured data type* defines objects that are data structures whose parts are objects of other data types, scalar or structured, even possibly of the same type.

stepwise data refinement

Structured data types permit stepwise refinement by successive replacement. In a refinement step, a scalar data type is replaced by a structured data type, introducing additional instances of scalar data types. The refinement process continues in this fashion until the data types of the programming language at hand have been reached. In parallel, the operations and tests of the original scalar data type are redefined in terms of more detailed operations and tests in the structured data type. For example, a matrix of complex numbers, regarded as a scalar data type in a high-level design, can be expanded to a pair of real numbers. These numbers are then expanded to a pair of integers (exponent, mantissa). At each step, operations and tests on the data are also reexpressed.

In addition to data design techniques, this practice also specifies expressive forms for defining data organization. The detailed organization of data is often expressed in natural language or graphic descriptions of formats, field layouts, word boundaries, etc. Data organization, however, can be expressed with greater clarity using mathematical techniques, such as formal grammars, regular expressions, and recursive formulas. These techniques emphasize hierarchical patterns in data organization, and provide a structural framework for the design of programs that process the data.

modular design

The modular design practice specifies a methodology for designing the synchronous logic of software systems. Modular design is the principal means for hierarchical decomposition and organization, once an overall hardware/software system design has been completed.[7,9] It makes use of techniques described in the program design and program design verification practices, and introduces two additional concepts: *state machines* and *modules*.

state machines and modules

Briefly, a state machine is a mathematical function that can be used to specify programs and data. A state machine m is defined in terms of input, output, states, and transitions, as follows:

m = {((input, state), (newstate, output))}

Each member of the set defines a transition from a current state and an input to a new state and an output (possibly null). In software terms, the state machine m corresponds to program operations on input and state data to produce new state data and output, where the data are regarded as *persistent*, that is, data that survive (i.e., stored) between program executions. A module is composed of a specification part and a design part. An *intended state machine* is the specification part of a module, just as an intended function is the specification part of a program. The design part of a module is normally composed of a single structured program paired with persistent data.

The use of modular design is intended to control complexity by organizing a design into a hierarchy of modules, where each module hides the implementation of data and operations from module users. Modular design also maintains data integrity by defining the persistent data of each module to correspond to a state of the intended state machine, and by permitting access to those data only through the module program. This also ensures completeness of the design. The intended state machine idea is a unifying concept that helps to determine that a collection of program operations should be grouped into a module and that all required operations on the data of the module have been defined. That is, the module carries out the correct data operations in every possible circumstance.

Modules result in reduced complexity in system design because they abstract out (or hide) details of representation, residency, and format of persistent data, and the algorithmic details of data processing. Because they provide an abstract view of data to their users, modules are also referred to as *data abstractions*.

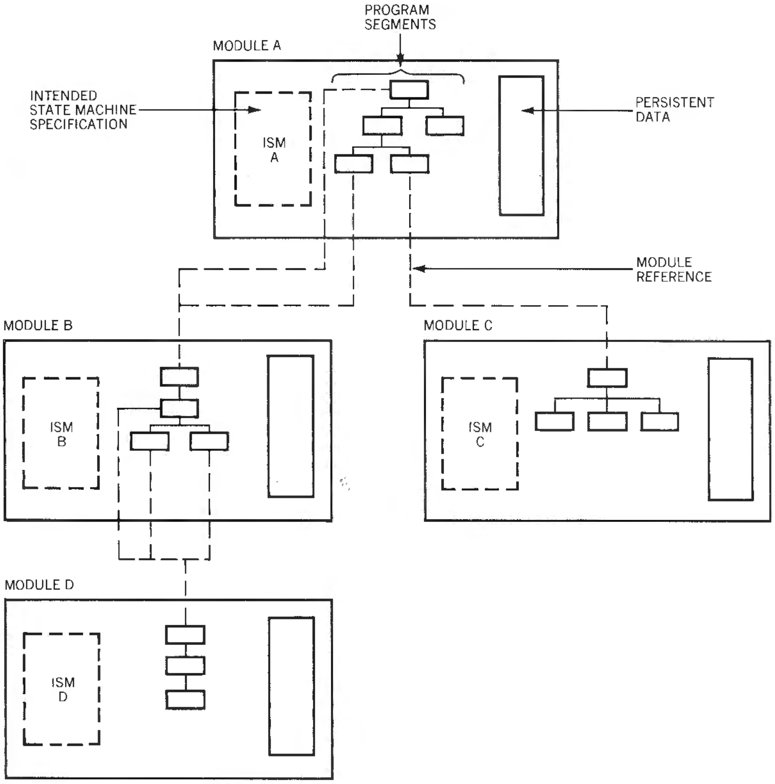
specification by intended state machines

An intended state machine is a precise specification for the function of a software system or system part, such as a subsystem or common service. Intended state machines can define services to module users (including definition of interfaces for invoking those services) at all levels of decomposition in a software system, without getting into details of program design and data organization and storage. For example, an entire synchronous text processing system can be specified in terms of an intended state machine, as can its individual subsystems, such as text update, text retrieval, file maintenance, etc., as well as each of its low-level common services, such as directory management, user status management, space allocation, etc.

module program

The module program is the sole interface for module users and provides the only permissible access to the persistent data of the module. The program may reference the module programs of other modules in carrying out its operations. (In implementation, a module containing multiple programs accessible by users may be a reasonable alternative, despite the complexity introduced by multiple interfaces.) A module program's inputs and outputs correspond to the inputs and outputs of the intended state machine. Its operations correspond to the state transitions, and its persistent data correspond to a state of the intended state machine.

Figure 2 A module-structured design



The persistence of data in a module-structured system ranges from permanent data base data in a resident module, which may survive indefinitely, to local state data of transient modules, which may survive only momentarily between successive invocations within an active job or task.

modular design by stepwise refinement

Modular design is carried out by *stepwise module refinement* of intended state machines and their designs. The process begins by describing an intended state machine, which is then elaborated as a module design consisting of a module program, persistent data, and possible services defined by additional intended state machines. The refinement continues in this manner until the lowest-level modules have been designed. This design process is a direct extension to stepwise refinement of intended functions into programs that may reference additional intended functions.

Specifically, the first step in a module design is the definition of its persistent data and the intended function of its program. Any abstract objects (such as sets) in the state of an intended state machine are elaborated into persistent data using data refinement techniques. The intended function is elaborated using stepwise program refinement techniques. In this process, opportunities may arise to organize data and operations into new intended state machines at a lower level, to be likewise implemented as modules. Note that during refinement, modules containing no persistent data may arise. For example, it makes sense to group scientific subroutine operations into a module, even though they typically reference no persistent data.

A module program undergoes stepwise refinement into a local hierarchy of program segments, any of which may run the programs of other modules to provide access to their persistent data. Thus, a module-structured system is composed of a hierarchy of modules with program refinements defining connections between levels in the hierarchy. Figure 2 depicts an imagined module hierarchy in graphic form.

correctness verification

The module defines a *module state machine* as all possible executions of its program on input and persistent data, just as a program defines a *program function* as all possible executions on input. A module is correct if its intended state machine is equivalent to (or a subset of) its module state machine. At each refinement step, a designer must demonstrate that this equivalence holds. Much of the effort in the proof involves proving that the module program correctly implements its intended function. This should be done by direct inspection if possible, otherwise by verification techniques of sufficient rigor, as described in the program design verification practice. If abstract data objects and operations are used in the intended state machine description and then refined into more complex data objects and operations in the module, correspondence between the levels must be demonstrated. Finally, it must be shown that the correct persistent data have been identified.

module implementation

Many operating systems and languages do not provide adequate implementation support for data abstraction by modules. For example, scope rules in many languages require that files for persistent data intended to be hidden in a module must actually be declared in a higher-level module.

Concluding remarks

The software design practices summarize technical principles for creating software system designs out of requirements. And they define a series of development checkpoints for technical management as well, in terms of specific intermediate work products along the way from requirements to design. These work products record a progression of reasoning and analysis that permits continual review and improvement of designs. The practices legitimize these work products and sanction their development. Each work product can be allocated and managed for cost and quality, so that the state of development is never in doubt.

Appendix A: Elements of the program design practice

Introduction

1.1 SCOPE

This practice specifies a function-based methodology for creating and recording a correct program design to satisfy a *specification function.*

1.2 OBJECTIVES

The use of the methodology is intended to reduce complexity and maintain intellectual manageability in program design. This is accomplished by designing programs to satisfy hierarchies of functions, thereby localizing design decisions and correctness demonstrations.

1.3 APPLICATION

This practice applies to all new program designs developed by FSD, including program designs appearing in requirements, specification, and design documentation, and stored in computer libraries.

1.4 AUTHORIZATION

This practice has been approved by the FSD Software Technology Steering Group and the FSD Standards Manager.

. . .

Practice

2.1 DESIGN METHODOLOGY

2.1.1 Responsibility. An individual will be assigned responsibility for the design of each program, whether that design is developed as an individual activity, or as a team effort.

2.1.2 Stepwise Refinement. Beginning with a specification function, a program design is created and recorded as a hierarchy of control structure expansions by the process of *stepwise refinement,* using the *Axiom of Replacement* as the only rule of construction. Data structures are also introduced in a hierarchical manner, to support the local operations of each refinement. The program design *segment* is a natural unit of refinement for each step. Stepwise refinement is not a mechanical process, and a good understanding of overall program and data structure is required before commencing segment design. The depth of design will vary with complexity; the refinement process should terminate at the point where further refinements become obvious.

2.1.3 Stepwise Reorganization. In complex design situations, the strategy of *stepwise reorganization* should be considered, to keep correctness arguments manageable by designing for function first, and reorganizing for efficiency later.

2.1.4. Correctness Verification. At each refinement step, the designer must be able to convince himself and others that the *program function* of the refinement is equivalent to its specification (or intended) function. The program function of the refinement may be self-evident, and the correctness determination made by direct inspection. If the program function is not self-evident, a simpler design should be considered; otherwise, correctness verification techniques with sufficient rigor to verify correctness must be applied.

2.1.5 Logical Commentary. The program function of every segment should be defined or referenced in a logical commentary *action comment*. Important intermediate program functions should also be defined, including those for program parts which have been informally or formally proven correct. *Status comments* should be included where appropriate, as well.

2.1.6 Design Modification. Program designs should be modified as necessary, both to correspond to redefinition of their specification functions, and to reflect any design improvements discovered in implementation. Verification considerations apply to all design modifications.

2.2 WORK PRODUCTS

Program designs in the documentation specified in section 1.3 should be developed using the methodology specified in section 2.1. Program designs should be kept current and always available for study and review.

Appendix B: A miniature segment-structured program design

```
proc oddeven(output, input) [set next consecutive output sequence members to 1 or 0 depending on whether corresponding input sequence members are odd or even]
   var input, output: sequence of integer
   var x: integer
   while
      input ≠ empty
   do [next(output) := oddeven(next(input))]
      x := next(input)
      if
         x > 0
      then [convert positive x odd or even into 1 or 0]
         run positive(x)
      else [convert nonpositive x odd or even into 1 or 0]
         run nonpositive(x)
```

```
        fi
        next(output) := x
    od
corp

proc positive(x) [convert positive x odd or even into 1 or 0]
    var x: integer
    while
        x > 1
    do
        x := x − 2
    od
corp

proc nonpositive(x) [convert nonpositive x odd or even
                     into 1 or 0]
    var x: integer
    while
        x < 0
    do
        x := x + 2
    od
corp
```

Appendix C: A structured program in three refinement steps

```
step 1:
        proc maxmin(x,y,t,n)
            var x,y,n: integer
            var t: array(1..n) of integer
            x,y := max(t(1:n)), min(t(1:n))
        corp

step 2:
        proc maxmin(x,y,t,n) [x,y := max(t(1:n)),min(t(1:n))]
            var x,y,n: integer
            var t: array(1..n) of integer
            x,y := t(1), t(1)
            [x,y := max(x,t(2:n)),min(y,t(2:n))]
            for
                i = 2 to n by 1
            do
                x,y := max(x,t(i)),min(y,t(i))
            od
        corp

step 3:
        proc maxmin(x,y,t,n) [x,y := max(t(1:n)),min(t(1:n))]
            var x,y,n: integer
            var t: array(1..n) of integer
            x,y := t(1), t(1)
```

```
    [x,y := max(x,t(2:n)), min(y,t(2:n))]
    for
       i = 2 to n by 1
    do [x,y := max(x,t(i)),min(y,t(i))]
       if
         t(i) > x
       then
         x := t(i)
       fi
       if
         t(i) < y
       then
         y := t(i)
       fi
    od
corp
```

Appendix D: A miniature correctness proof

A. Intended function

(f) $x,y := y,x$

B. Program

(S1) $x := x + y$

(S2) $y := x - y$

(S3) $x := x - y$

C. Proof

trace table:

row	assignment	x	y
1	$x := x + y$	$x_1 = x_0 + y_0$	$y_1 = y_0$
2	$y := x - y$	$x_2 = x_1$	$y_2 = x_1 - y_1$
3	$x := x - y$	$x_3 = x_2 - y_2$	$y_3 = y_2$

derivations:

$x_3 = x_2 - y_2$
$x_3 = x_1 - (x_1 - y_1)$
$x_3 = y_1$
$x_3 = y_0$

$y_3 = y_2$
$y_3 = x_1 - y_1$
$y_3 = x_0 + y_0 - y_0$
$y_3 = x_0$

Therefore
$p = (x,y := y,x) = f$

pass

The author is located at the IBM Federal Systems Division, 10215 Fernwood Road, Bethesda, MD 20034.

The management of software engineering Part IV: Software development practices

by M. Dyer

The IBM Federal Systems Division began a continuing search for new and better software development methods in the early 1950s when it was participating in the SAGE air defense system. Since then, members of FSD have been developing large, complex, real-time systems exemplified by the manned spacecraft projects Mercury, Gemini, Apollo, and the Space Shuttle. In such projects, military and civilian, software development is characterized by challenging targets and severe constraints. Schedules are tight, workloads are heavy, computer processing must fit within restrictive time slices and memory allocations; yet results are to be error-free. Added to these stringent requirements is the need to minimize cost but still make the system robust enough to be operated and maintained by someone other than the developer.

This experience motivated the merger of things learned on-the-job with advances in the discipline of software engineering. The program that evolved covers design, development, and management with the objective of intellectual control of the software engineering process.

In this paper on software development, the focus is on the blend of modern software methods with established development practices. Reducing diversity, increasing visibility, and improving productivity in the development process are the principal means of intellectual control of development. Improved product quality, product transportability, and product adaptability are longer-range goals.

The development methodology is defined in terms of practices that recognize the increased precision introduced by modern design methods and that attempt to introduce the rigor of modern design into the methods of software product development. Code management practices deal with the implementation of software and the control of its release as a product. Integration engineering practices address plans for building software products.

Code management

Contemporary software development methods reflect modern programming technology. Structured programming techniques, employed with high-order programming languages, are *de facto* standards. The prominence of programming support libraries, with features to support configuration management and quality assurance functions, and the growing acceptance of top-down design methods, program design languages, and design review techniques, are further evidence of new technology acceptance.

The most effective procedures used within FSD form the basis of code management practices that support the development of software products. These tested methods aim toward setting a minimum standard for software development in the following categories:

- Programming language.
- Coding standards and conventions.
- Computer product support software.
- Hierarchical program control library.
- Software development environment.

programming language

The first three categories influence the implementation of the software, whereas the latter two focus on the packaging of the software into a deliverable product.

Software products should be implemented with High-Order Programming Languages (HOLs) that simplify the translation of design specifications—as documented in a design language—into code. It is desirable that the syntax of the programming language include control and data structures and be consistent with the design language syntax. In the comprehensive FSD software engineering program, a Process Design Language (PDL) is recommended; however, no single high-order programming language has been specified. The reason for such latitude is that FSD customers often require their contractor to use a language that is both appropriate for the customer's problem environment and familiar to the customer's programmers. Thus, the programming language practice identifies for Department of Defense (DOD) applications languages such as FORTRAN, COBOL, JOVIAL, etc. For National Aeronautics and Space Administration (NASA) applications, the HAL/S language is identified. For internal IBM applications, the PL/I, PL/S, and APL languages are identified. Programmers are advised to use one high-order language per project, which should be selected from the set of HOLs listed in the practice.

System designs, documented with a design language, are entered in a program support library. The selection of the HOL is influenced by its consistency with the design language. To extend the list of qualified HOLs, consistency need not be provided directly by the HOL; it can be provided by a preprocessor in the program support library.

Table 1 Language recommendations for classes of software products

Language recommendation	*Class of software product*
High-order language (HOL)	Program development and generation Compiler/assembler Link editor/loader Utilities Library support Data reduction Applications
HOL with assembly assist	Program development and generation Hardware simulation System simulation Diagnostics
Assembly with HOL elements	Executive
Assembly	Data recording/measurement Microcode

In general, programmers are advised to restrict their use of assembly language to those portions of a software product involving critical time or space constraints (and to those products implemented for processors that have only assembly language support). The recommendations of the practice as of 1980 for various classes of software products are shown in Table 1.

coding standards and conventions

Other decisions to be made prior to software implementation dealing with project standards and conventions are the following:

- Standards for writing code.
- Standard interfaces with operating system software.
- Conventions for using a Program Support Library (PSL) system to control the product development and obtain visibility into the development process.
- Conventions for packaging code into controllable objects.

Standards for written code include rules for naming program and data variables and rules regarding program commentary. Symbol names are intended to improve the documentation of software and ensure code readability. Commentary covers traditional prologues and statement comments as well as the logical commentary that evolves during the design process. Good commentary makes a program intelligible to persons other than the author, including operations personnel.

To support configuration management goals the coding practice discusses the use of alphanumeric statement identifiers. These identifiers permit the inclusion of version number, revision level within version, and standard statement sequence numbers that have proved valuable in the control of software products that change with time.

Software development assumes the use of executive software in the typical project environment for which interface conventions must be established. Initialization/termination, interrupt handling, resource allocation and management, and input/output device handling are the minimum functions to be handled by executive software. Coding these functions is both difficult and time-consuming. The purpose of standards in this area is to introduce consistency in using the executive.

Program Support Library (PSL) systems typically maintain source statements in both the design and the programmming language and provide linkage to executive software for compilation and execution. The PSL system may provide language preprocessors for structured language forms, as necessary. Through the PSL system, the user is supported in interactive, batch, and dedicated development environments. Conventions for using a PSL system provide visibility by identifying the requirements for collecting and reporting status information, such as segment type identification, number of source statements, number of source statement updates, date of last update, and current version and revision level.

The coding practice also defines conventions for packaging code into products, considering execution time addressability and the packaging requirements of peripheral storage devices. A segment of code implements a unit of function; a segment may range up to fifty lines in length, but should not exceed a page. Transportability considerations suggest that programs and data be designed to be relocatable to any area in main memory for execution without requiring any knowledge of absolute addresses. Data files designed for storage on peripheral input/output devices are organized in logical records and require no knowledge of the physical structures for storage devices.

computer product support software

Within the FSD business environment, software is routinely developed for special noncommercial machines (some of which are FSD hardware products) with limited or no support software. The intent of the computer product software support practice is to establish the minimum levels of support software that should be available or developed for these classes of machines. The practice separates computer products into data processing systems, central processing units, peripheral storage devices, and terminal devices. The minimum levels of support software that should be developed and maintained as part of the hardware development process include the following:

- *Terminal device* software supports decoding of keyboard input entries, the generalization of the input data into standard message formats, and the notification of input message availability. For the output side, the software uses standard message formats for identifying output data, performs data encoding for symbol generation, graphics generation, and display control, handles the physical transmission of data, and monitors transmission status.

- *Peripheral storage device* software handles the transmission of data to and from a central processing unit and storage devices, supports the definition and use of logical storage units (files and records) that are function-dependent (as opposed to device-dependent), processes device controls (e.g., end of tape), and monitors transmission status.
- *Processing unit* software handles the identification and processing of execution interrupts and the allocation and scheduling of the central processing unit resources.
- *Data processing system* software supports the initialization, termination, and use of all computer products in the configuration. It also provides Program Support Library (PSL) facilities, language processors, linkage editor functions, and software simulations of computer products.

By including these minimum capabilities in every hardware system, a base exists on which the software engineering program can build.

hierarchical programming control library

Programming Support Library (PSL) systems have been widely adopted as productivity aids for the programmer. The PSL automates the processes of code capture, retention, and retrieval, as well as program linkage, compilation, and execution, and code modification and output listing. The same PSL can provide important assistance in development control by segregating project components that are complete from those in progress. The hierarchical programming control practice identifies the need for a library structure with at least three levels and for library procedures that permit users to do the following:

- Realize the productivity benefits of the PSL.
- Promote programs from one level to the next.
- Build program products by combining PSL entries.
- Maintain source code integrity during checkout and integration.
- Support software quality assurance functions.
- Support software configuration management functions.

The levels of PSL should bear a hierarchical relationship to each other and include the following as a minimum:

- *Development level.* Programs under development, or testing by the software implementer enter PSL at this, the lowest, level of the hierarchy. The implementer interacts directly with his own code as filed under his identifier. Development level code is seldom useful to others and may be accessible only to its author.

- *Integration level.* This level contains developed programs, fully debugged by their authors, ready to be integrated with other programs and tested as components of a software product. Programs are promoted from the development level to the integration level; integrated, checked-out software packages are promoted to the release level.
- *Release level.* Software ready for delivery to the customer is stored at the release level. In some cases, users can execute the code to obtain operational results; however, it is more likely that users obtain a copy of the release level software product and run it independently of the PSL, although the PSL remains the source of the master copy of the latest version of the software product.

When a user refers to a level of the PSL, he can expect to find current, approved data. That is, the development level contains today's version of the implementer's work; the integration level contains only debugged programs; the release level contains the version authorized for release to customers. PSL procedures are designed to deliver what the user expects—a single copy of data commensurate with development status. At the same time, the PSL may support multiple copies and additional levels. Such flexibility facilitates fallback; it supports multiple releases to different users or for different purposes; it permits demotion of programs undergoing modification while retaining a useful earlier version at higher levels; and, in general, flexibility protects the integrity of the library contents at each hierarchical level.

A request should automatically result in a response from a standard library level. As an option, however, the access mechanism should allow an authorized user to select data from other levels. Authorization control, which governs who can read, write, or modify library entries, is provided by the access mechanism, possibly using a password technique.

As a rule, customer delivery of software products is made using source code data. This procedure results in products that can be created from approved source code (i.e., free of machine language fixes or patches). Customer or contractual requirements may dictate the release of products containing patches, but these should be considered as exceptional cases. In such cases, manual control procedures should be used to manage patches in the released software, and normal configuration control procedures should be executed in parallel to ensure source-level integrity of the released software.

software development environment

Because of the diversity of the customer set within FSD, different development environments have evolved to meet individual needs. A minimum set of development procedures have been identified as applicable to the various environments.

Table 2 Recommended development tool usage

Activity	*Interactive*	*Batch*	*Dedicated*
Library organization/setup		●	
Design language input/edit	●		
Programming language input/edit	●		
Test case input/edit	●		
Compilation/assembly			
Up to 1000 statements	●		
Greater than 1000 statements		●	
Program link edits	●		
Unit test execution			
User test data	●		
Simulation controlled		●	
Hierarchical programming control library parameter input/edit	●		
Hierarchical programming control library generation		●	
Software integration testing			●
Software/hardware integration testing			●
Integration test data reduction		●	
Status report generation			
Queries	●		
Reports		●	

For all aspects of software development—from design through product release—the use of interactive terminals is encouraged. Batch processing, with its average twenty-four-hour turnaround, is restricted to the execution of production programs, where possible. Dedicated operations, where an entire machine is turned over to one programmer or test team, is similarly limited, specifically to integration activities involving specialized hardware requiring computer system reconfiguration. The software development environment practices are summarized in Table 2, which shows how the guidance is broken down by type of activity.

Integration engineering

Integration engineering has emerged as a new methodology, with roots in advanced software design concepts. Therefore, integration engineering practices have been organized that support the phased integration of software and make integration planning an integral part of the modular design process. Integration engineering encourages the use of the modular design techniques of stepwise refinement and state machine hierarchical descriptions to detail the integration process and manage the specification of system interfaces. These practices also influence the software design process by introducing the ideas of incremental software development and establishing criteria for partitioning the development process to support phased integration. The integration engineering practices have been used to integrate software with software and software with hardware. The following four practices have been defined:

- Incremental software development.
- Software interface specification management.
- Software integration methodology.
- Simulation software.

As a group, these practices govern how a large scope of effort is broken into manageable parts, how the parts are interconnected, how they are reintegrated into a software product, and how—through simulation—the process is controlled throughout the life cycle.

In any activity where the job to be done is too large for one person to handle, it is necessary to break the job apart. The very act of partitioning the system introduces development process problems because interactive components are more complex than single entities. Integration engineering addresses the plans for partitioning in such a way that the pieces can be developed independently yet come together at the right time to fit software, hardware, and system integration schedules. Simulation is emphasized since it permits evaluation of the incomplete, developing system using simulated components in place of the missing, real components.

incremental software development

The development of software in increments is a key integration engineering concept. The incremental software development practice provides guidelines for developing software products in increments, for selecting the number of increments, and for determining the capabilities needed in each increment to support integration. Software is partitioned into increments, whose development is scheduled or phased over the total development cycle. Each increment is a subset of the planned software product, and provides a specified system function(s). As a minimum, partitioning should satisfy the following requirements:

- Be natural or logical with respect to the operational system or application.
- Organize each increment to maximize the separation of its function(s) from function(s) in other increments.
- Structure the phasing of increment development to minimize modification of previously completed increments due to the implementation of subsequent increments.

Partitioning is addressed in the software specification and design process so that increments and their development schedules can be managed to protect against project schedule erosion.

As a guideline for a top-down integration strategy, phased integration should be supported by the following four software increments that would be developed in the indicated sequence:

1. *Initial increment*—exercises all interfaces with operating system software; includes selected processing kernels that represent high-risk, system-critical functions.
2. *Intermediate increment*—exercises explicit interface specifications.
3. *Interim increment*—exercises selected system function(s), depending on application complexity. Multiple interim increments may be required first to exercise critical (prime system) functions and subsequently to exercise secondary functions.
4. *Final increment*—exercises total system function.

Alternate integration strategies would be based on variations of this top-down strategy, wherein the role of the intermediate increment has lesser significance. A *functional integration* strategy, where major system capabilities are organized into increments and integrated in successive phases, exercises only those interfaces that are significant to a specific functional capability, at any given phase. A *processing flow integration* strategy similarly addresses only subsets of the total interfaces during a given integration phase.

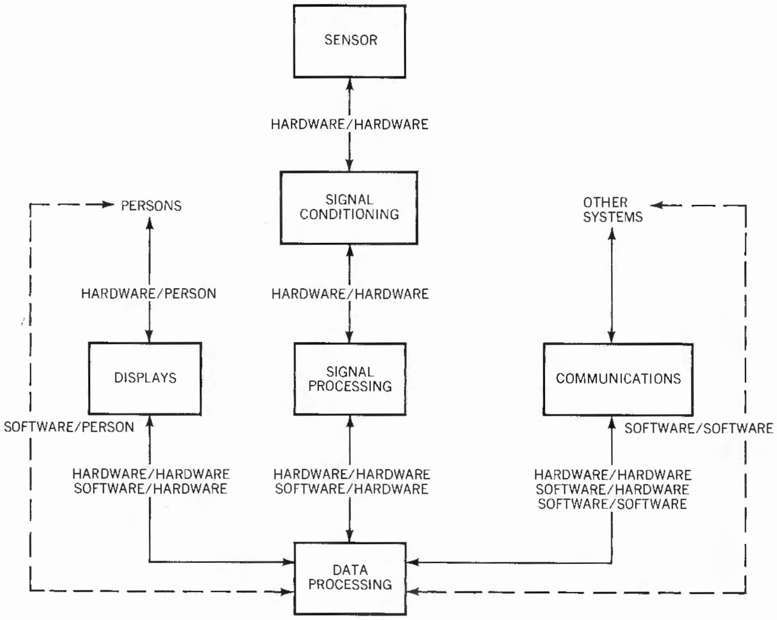
Data recording is a key element of a software system design and is incorporated in a manner that minimizes interference and distortion. The software for each increment is instrumented for measurement of such system resources as prime and secondary storage utilization. The measurements should be performed as part of the standard integration activity. Instrumentation that permits interfacing with simulations of missing hardware/software function is also included as required. The PSL system can support this instrumentation requirement with the use of program "stubs."

Data recording capabilities implemented to support testing should also be employed for operational data recording where possible. Technical performance estimates can then be accompanied by actual performance measurements. As these actual performance measurements become available, software simulations that may have been initialized with estimates should be continually calibrated to enhance their fidelity.

software interface specification management

Specification and control of interfaces is required for effective system development. Figure 1 indicates the potential interfaces found in systems that are typical of the FSD business area. The interface specification practice establishes criteria for managing interfaces for any of the following conditions:

Figure 1 Software system interfaces



- The interfacing elements are different in type (software, hardware, or person).
- The hardware and software controlling the interface are under concurrent development.
- The hardware and software controlling the interface are separately developed, whether for contractual, geographical, or organizational reasons.

The detailed data include an interface specification determined through stepwise refinement as part of software design. These specifications are recorded and controlled, either as separate documents or as part of the software specification. They contain descriptions of the external appearance and procedural protocols of each participant at an interface. The specification can cover connector layouts, signal levels, functions available at the interface, and rules for making contact and invoking functions across the interface. As a minimum, the following interfaces should be specified:

- Interfaces between software and hardware:

  Interfaces between support software and computer products, such as processors, when these products are part of the development effort.

The programmable instruction set (whether hardwired or microprogrammed) for the selected central processing unit, as normally documented in a principles of operation manual.

Interfaces with application-specific hardware that is part of the system under development.

- Interfaces between two software products:

  Interfaces between software under development and existing support software products, such as operating systems whose use is planned for the system development.

  Interfaces between software products that are physically separated in different processors and logically connected through an intercomputer channel mechanism.

  Interface with shared system-level data structures. This interface is of critical significance with distributed software architecture.

- Interfaces between a software product and the person using it. The interfaces between the software and intended system users normally involve expansion and clarification of an established software/hardware or software/software specification.

Given an incremental development plan and a well-defined set of system interfaces, integration can proceed smoothly, without the delays that are caused when components fail to fit together.

software integration methodology

Integration is a controlled process by which software increments are integrated in environments that—at successive integration phases—more fully approximate the intended software system function. Effective control requires planning, design consideration, and product management. Though the emphasis is on software integration, the methods are equally applicable to a larger system environment that includes the integration of software and hardware components.

Planning for software integration should be initiated as part of the software design activity and should support the development of software specifications. These specifications record the systematic refinement of software requirements to the program level and are based on documented system-level requirements. Integration considerations are factored into the software design so that the software design supports the partitioning rules for incremental development. Specifically, the design reflects a separation of system function(s) that can be comprehensively tested and that permits the structuring of integration increments. The design also permits the testing of all specified system requirements. The specification of the software functions identifies the system requirement(s) to be tested. In addition, the identified inputs and outputs represent a basis for preparing test plans.

Software integration plans are recorded in controlled documents containing the following minimum information:

- Scheduled phasing of the integration increments.
- System functions included in each increment.
- Test plans to be executed for each increment with an assessment of the test coverage for the system functions embodied in the increment. (The successful execution of these test plans defines the exit condition from integration.)
- Support requirements for each increment in terms of system hardware simulation, tools, and project resources.
- Criteria for demonstrating that the increment is ready for integration. These criteria, a subset of the test plan for the increment, define the exit condition from the unit test.
- Quality assurance plans for the tracking and follow-up of errors discovered during the integration process.

Software integration plans should take account of total system integration and test plans and organize increments to support the system-level planning requirements. This is particularly important in major systems developments involving significant numbers of hardware and software elements. In such developments, hardware plans identify the separate integration and test of hardware, using software diagnostic tools, prior to the integration of hardware with system software. Incremental development of the system software can support the phased integration of a total system by providing subsets of the system software to assist in the total system integration.

Procedures that define the integration process at each increment are developed using refinement techniques that are conducted in parallel with the stepwise refinement of the software design. The procedures document the results of detailing test plans into a hierarchy of test cases to be executed during the integration activity.

When multiple functions must be included in a single integration increment, a stepwise integration within the increment is performed. Functions are integrated, one at a time, building on the existing stable base with single functions tested independently for dependability and readiness. The concept of dependability requires careful control of modifications to functions during integration. Modifications in response to problems found during the integration testing process must be made. However, modification for function growth—as directed by approved Engineering Change Proposals (ECPs)—should be phased into subsequent integration increments.

Table 3 Primary roles of simulation software

Stage in life cycle	*Type of simulation software*				
	Processor	*Interface*	*Environment*	*Computer system*	*Application system*
System definition				Requirements allocation analysis	Concept formulation analysis
Software design				Design tradeoff analysis	
Software development	Unit test support			Design control analysis	
Software system test		Test and integration support	Test and integration support		
System/ acceptance test		Acceptance test support	Acceptance test support		
Operations and maintenance		Training and maintenance support	Training and maintenance support	Design change analysis	Formulative design change analysis

The Program Support Library (PSL) system provides facilities for the storage of test-case libraries and for the segregation of software elements included in an integration increment. A group separate from the software developers should have responsibility for planning the software integration process, for developing the integration procedures, and for integrating the software according to these procedures.

simulation software

Simulation can be effective in several ways in most software developments. In the early stages, when little actual software exists, simulation by analytical methods can be used to evaluate designs and check algorithms. Later, as working code becomes available, simulators can supplement it to support system tests. After release, simulation is still helpful in training and as an updating aid. Various support roles for simulation software are listed in Table 3. Five types of simulation software are shown with their primary roles arranged in life-cycle sequence. The simulation software practice recommends that simulation be used for the indicated purposes to the extent justified by the nature, size, and budget of the project.

Processor simulation permits software development to proceed independently of processor development. The simulator consists of software representing the instruction-level operations of the proposed processor. Support services, including dumps, snapshots, traces, and timing routines are normally provided.

Interface simulation permits parallel development of the major components of a system—hardware, software, system operators. The simulator is software or hardware representing the behavior of each component when its functions are invoked. It can be used to provide responses expected from missing components and to verify the correct implementation of interface protocols.

Environmental simulation provides controlled conditions in which to develop and check out systems under development. The simulator represents the functional behavior of the hardware, software, and operational environment external to the system under development. It is usually implemented as software and run on a separate machine from the development software. The separate machine can, of course, be a real machine or a virtual machine. During a simulation, the environment can be represented by function responses as in an interface simulation or it can be set up as a script to drive a set of tests. In the latter mode, for example, a traffic control software system could be driven by a script that supplies traffic slowly to test basic functions, faster to test real-time performance, and still faster to test peak-load or overload behavior.

Computer system simulation, as defined in the simulation software practice, is an aid to decision-makers concerned with the effect of a design change on a complex system. Mathematical models are used to represent computer system resources and their utilization in terms of program path lengths, memory allocation, disk accesses, etc., as defined by the software design for a given operational scenario. Initial designs are modeled at a fairly gross level. As the design matures, the models get more precise. At each stage, the models support tradeoff analyses of alternative design decisions. After the first release of a software product, the same modeling approach can be used with performance measurements to obtain quite precise evaluations of design change proposals.

Application system simulation uses software to simulate a physical process associated with an application problem for which a system solution may or may not be implemented. The simulation is actually an alternative to actual system development. Simulations to determine the feasibility of a system concept or the requirements for a proposed system solution are typical of this simulation type.

Concluding remarks

The IBM Federal Systems Division has pioneered the development of large-scale, complex software products for various government agencies. The software engineering program has attempted to blend the best aspects of this unique experience with evolving software technologies to establish a set of uniform software development practices. These practices include code management activities for software implementation that promote the use of uniform, consistent techniques and tools for improved productivity and quality. These practices also address integration engineering activities for software product development and focus on the control of the most difficult aspect of software development—the coordination of independently developed, closely related, complex elements. Control is achieved by careful system partitioning, incremental product construction, and constant product evaluation. Overall, the FSD software development practices stress product visibility, dependable tools, easily understood procedures, and positive feedback at project checkpoints. The practical result of this approach has been an increase in the manageability of FSD contracts.

The author is located at the IBM Federal Systems Division, 10215 Fernwood Road, Bethesda, MD 20034.

The management of software engineering Part V: Software engineering management practices

by R. E. Quinnan

The IBM Federal Systems Division software engineering program was organized to support the design and development of software products. This program includes design practices that deal with the systematic decomposition of software designs into hierarchically related programs. This procedure results in products with structural integrity that are easy to use, maintain, and adapt. Development practices in the software engineering program deal with software implementation and integration engineering. The discipline of management practices closes and strengthens the triangle model of software engineering.

This part of our paper focuses on these management practices and discusses the plans and controls they provide to monitor progress and performance during the software life cycle. The software engineering management practices reflect the experiences of successful management teams and are familiar to most software managers. However, the effectiveness of these practices is significantly improved when the associated design and development practices are implemented. Uniformity and consistency resulting from good design and orderly development underlie the predictability and responsiveness of the management practices.

Software engineering management model

Our software engineering management model is composed of three sets of practices: (1) technical review; (2) cost management; and (3) software program management. The technical reviews are conducted during the development of a software product at specified checkpoints and for well-defined purposes. Cost management prescribes a method of planning, estimating, measuring, and controlling a developing software product to meet a cost objective. The software program management practice establishes a project environment and management relationships that foster complete, precise, and efficient communications within and between groups. The model is based on the software life-cycle activities described by O'Neill in Part II of this paper.

Table 1 Software life cycle

Activity	*Work components*	*Outputs*
System definition	Software requirements definition	Software system requirements specifications
	Software system description	Software system description document
	Software development planning	Software development plan
	Engineering change analysis	Engineering change proposals
Software design	Functional design	Functional design specifications
	Program design	Program design specifications
	Test design	Test design specification
	Software tools	Utilities, debugging aids
	Design evaluation	
Software development	Module development	Development (module) libraries
	Development testing	Development test procedures/ reports
	Problem analysis and correction	Program modifications
Software system test	Software system test procedures	Software system test procedures
	Software integration and test	Software system integration library, test reports
System/ acceptance test	System test support	System library, test reports
	Acceptance test support	Delivered software system/ acceptance library
Operational support	System operation support	Level of effort assistance, maintenance
	Training	Level of effort training manuals, courses
	Site deployment support	Level of effort assistance
General support	Project management	Level of effort
	Configuration management/ control	Procedures, standards, library control
	Software cost engineering	Cost management practice support
	Quality assurance	Audits, quality assurance plans
	Administration centers/technical publications	Level of effort

Each of these activities is in turn composed of the set of work components shown in Table 1, which identify the work performed, the expected end products, and criteria for completion. In a typical project, these activities and components overlap; baseline releases are defined to indicate when a component output has satisfied its completion criteria.

FSD projects can cover complete software product life cycles from concept formulation through end-of-life. Quite often, however, our responsibility spans system definition through acceptance test with a limited responsibility for operational support. The customer does his own requirements definition and runs his own operations in these cases. There are also some projects, usually large ones, in which several software subcontractors share

Table 2 Technical reviews within the software life cycle

Software life-cycle activities	*Related work components*	*Technical reviews*
System definition	Software requirements definition Software system description	System requirements Software system specification Test plan
	Software development planning	Documentation outline
Software design	Functional design	Software system design Integration plan
	Program design	Module design
	Test design	Test plan Test specification
	Software tools	
	Design evaluation	
Software development	Module development	Module implementation Unit test procedure
	Development testing	Module qualification Test procedure
Software system test	Software system test procedures System integration and test	Test procedures Software system qualification
System and acceptance test	System test support Acceptance test support	Software system acceptance Documentation completion
Operational support	System operation support Training Site deployment support	

the workload. Our role in such situations can span any or all of the life-cycle activities. Thus the life-cycle model provides a standard for determining the scope of software engineering in a particular project and serves as an improved communications method. The latter is especially important on larger projects where there is significant interaction between software engineering and other functional groups, such as program management and systems engineering.

technical reviews

It is well known that early detection of problems and errors is the most cost-effective method of quality control.[30] Since the person who generates a problem or an error can easily overlook it, we rely on technical reviews for thoroughness. This procedure brings the talent of a wider group of people to bear on each work product, quickly and efficiently.

Technical reviews develop a strategy within the software life cycle that permits the assessment and control of software activity. The strategy is associated with the life cycle just mentioned so that the technical results of each activity can be evaluated. Table 2 illustrates the correspondence of the technical reviews with the software life-cycle activities and the resulting work components. The number of actual reviews may vary, depending on individual project characteristics. Some reviews reoccur each time an event occurs, such as completion of a document outline. Multiple reviews can be conducted at one time; documentation, test, and integration reviews are normally conducted in conjunction with other reviews. In some cases, it may be convenient to run a review of nontechnical issues, such as contract compliance, budget tracking, and resource plans, immediately before or after a technical review. By such scheduling, technical problems may be completely resolved then and there by lining up all the resources and administrative approvals that might be needed.

Each review in Table 2 has a stated purpose outlined as follows:

- *System requirements.* Determine that software requirements for system capability are completely and correctly stated to permit development and use by the planned system user.
- *Software system specification.* Determine that software system specifications are complete and correct; ensure that each requirement for system capability can be traced through to the delivered software product.
- *Test plan.* Determine that the implementation of the software system is tested against the software system specification and that all requirements are checked out.
- *Documentation outline.* Determine that the outline for any planned document satisfies its objectives.
- *Software system design.* Determine that module designs comply with the software system specification and collectively implement the software system specification.
- *Integration plan.* Verify that there is a systematic approach to the implementation and testing of the software system.
- *Module design.* Determine that program and data designs comply with their module designs and implement their intended function.
- *Test specification.* Verify that test methods and materials comply with an approved test plan; evaluate functional and performance details of the tests versus the test objectives.
- *Module.* Verify that programs and modules are correctly implemented in accordance with their design and that unit test procedures have been established.
- *Test procedure.* Verify that test methods and materials comply with an approved test specification; evaluate the test operational scenario and machine execution control details.
- *Module qualification.* Determine that the module complies with the software specification; certify the module so that the code can be promoted from the project development library to the integration library.

- *Software system qualification.* Determine that the implemented software system complies with the software system specification; certify the system so that the code can be promoted from the integration library to the release library.
- *Software system acceptance.* Verify that the software system complies with all project deliverable objectives; certify the system so that the code for the software can be released to the customer (or the integration activity in a hardware/software system project).
- *Documentation completion.* Verify that the completed documentation satisfies its objectives and complies with its approved outline.

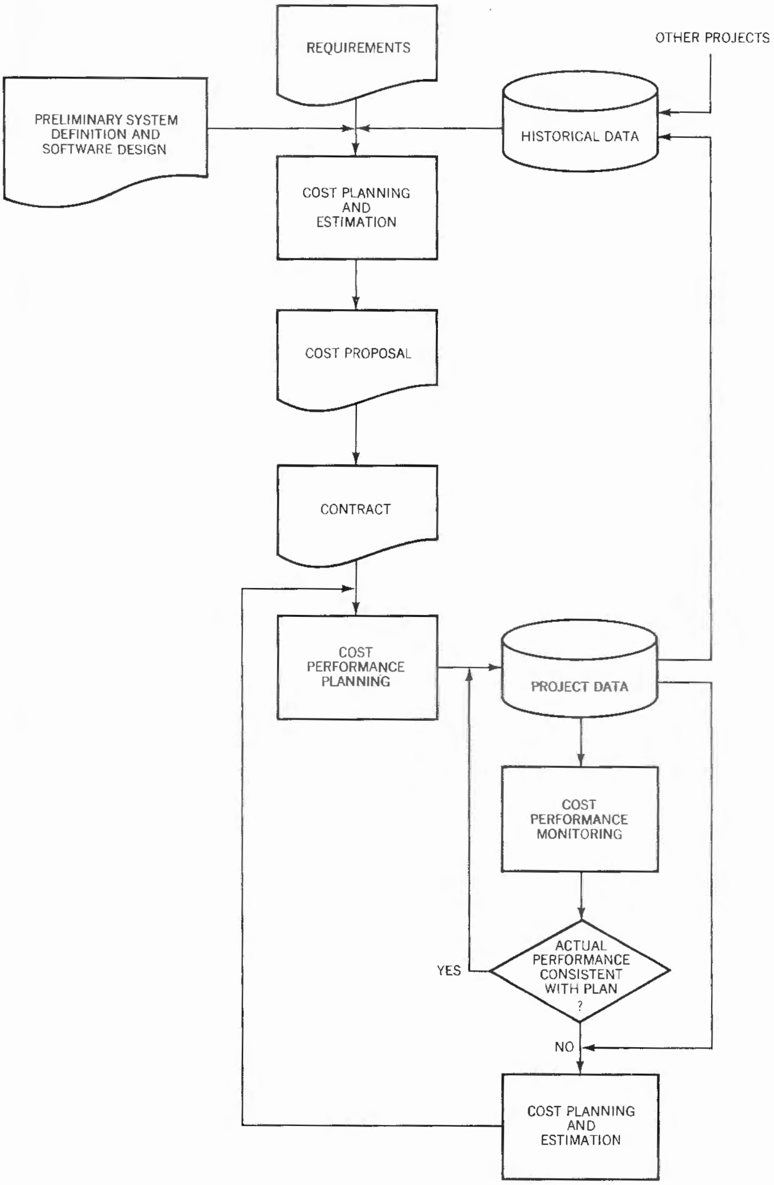
All these reviews are tools for project managers to use in assessing how well objectives are being met. To a large degree, the reviews deal with documents—specifications, test plans, test reports, procedure descriptions, and, ultimately, code. Reviewers can read the documents to assess the content and quality; they can talk to the developers to assess the intent of the implementation and to clarify unclear statements. Intuition and judgment are required, besides technical knowledge. Reviewers must spot weaknesses in the work products, propose fixes, and establish criteria for subsequent reviews to verify that the fixes are successful. The final acceptance reviews certify that we are confident that the software product is ready for the customer and can pass his acceptance test.

cost management

Software systems are built to provide specific capabilities for the user. Inevitably, the capabilities delivered depend on how much the user can afford to spend. In government contracts, as in business information systems, value/cost tradeoffs reflected in project budgets place constraints on software development plans. Our goal is to give our managers planning and control procedures that permit them to manage technical progress and cost at the same time.

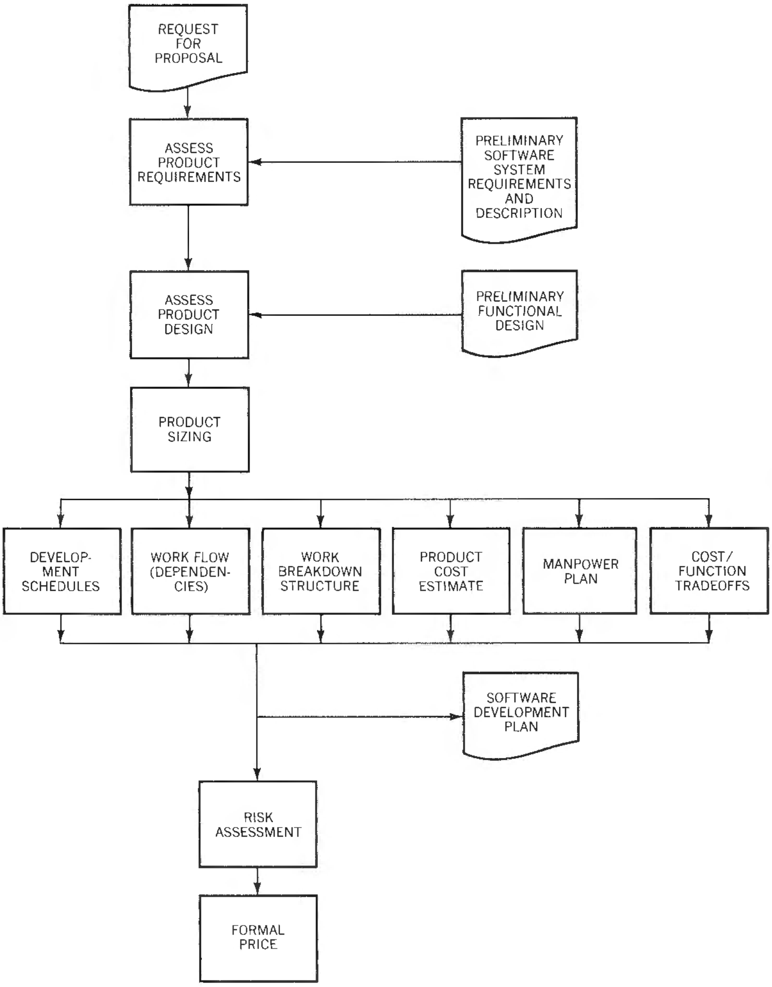
The cost management process, illustrated in Figure 1, starts when the software design is sufficiently detailed to support cost estimates and identify areas of risk. The planning and estimation steps are depicted in Figure 2. By spelling out all these steps in the cost management practice, we tend to avoid oversights. Looking back to 1964, when many large projects were severely underestimated, we now see that simple oversights can be identified as major sources of error.[31] Estimating methodology at that time focused only on actual programming activity; technical support, administration, and management were routinely omitted. Since the omissions were large—typically 150 percent of the direct programming—cost target misses were large. The solution for many years was to adjust the basic programming estimate by a factor that compensated for omissions. Meanwhile, an effort was

Figure 1 Cost management process


REQUIREMENTS
OTHER PROJECTS
PRELIMINARY SYSTEM DEFINITION AND SOFTWARE DESIGN
HISTORICAL DATA
COST PLANNING AND ESTIMATION
COST PROPOSAL
CONTRACT
COST PERFORMANCE PLANNING
PROJECT DATA
COST PERFORMANCE MONITORING
ACTUAL PERFORMANCE CONSISTENT WITH PLAN ?
YES
NO
COST PLANNING AND ESTIMATION

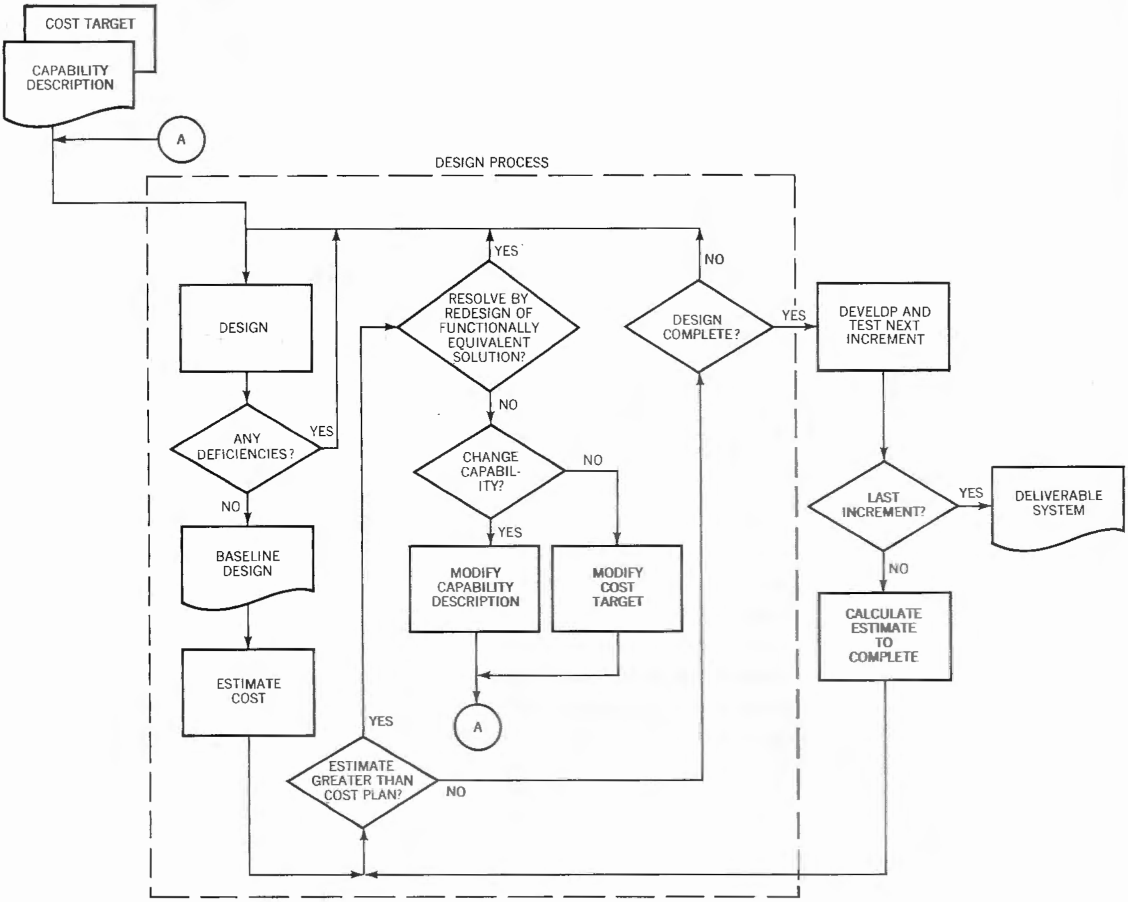
made to collect enough historical project data to replace the rule-of-thumb adjustment with more reliable methods. Now, we can tell our managers how to estimate each stage of the life cycle and how to deal explicitly with overhead support activities.[32] Our cost management practice, moreover, reminds managers to carry out each step without omissions. As a result, our proposals to customers contain a cost plan that can be tracked throughout the life cycle.

Figure 2 Planning and estimating



The cost plan contained in the proposal is refined further after contract award. At this time, detailed budgets are prepared and checkpoints are established. Procedures are also set up to collect performance measurements to permit an assessment of progress versus plan at checkpoints. The challenge in this process is that of obtaining a direct link between costs and technical progress. The fact that the expected amount of money has been spent by a given date does not necessarily indicate that the project is on schedule. Here again, our emphasis on tying the cost plan to the activities of the life cycle helps us. Expenditures are not merely projected on a month-by-month basis; they are related to specific work components and completion dates. Thus, reviews assess cost status, technical status, and expected cost status for the given technical status. A variance between actual results and expected results indicates potential problems or areas for improvement.

Figure 3 Design-to-cost



Modifications or growth in the estimated scope of work, deviations in expected productivity, or erroneous initial assumptions may require another cost planning and estimation cycle. For each iteration, the planning data are retained in the project data file for subsequent cost performance monitoring. The life-cycle model provides a checklist for assessing all the implications of changes, and the project data file provides actual performance data for cost planning and estimation.

design-to-cost

Cost management, as described, yields valid cost plans linked to technical performance. Our practice carries cost management farther by introducing design-to-cost guidance. Design, development, and management practices are applied in an integrated way to ensure that software technical management is consistent with cost management. The method, illustrated in Figure 3, consists of developing a design, estimating its cost, and ensuring that the design is cost-effective. To do this, design-to-cost goals are established, based on an understanding of the capabilities of the software and the related design solution. Plans to achieve these goals are developed by allocating costs to particular work components of the software system life cycle.

Design is an iterative process in which each design level is a refinement of the previous level. At each stage, design and cost alternatives are examined. Those that best satisfy the project objectives are prepared for review and selection by the project sponsor. If no alternative fits the cost target, several courses of action are available. The most common one is to go back to the designers and ask for a less costly, and perhaps less attractive, design. If the target has been missed by a large amount—and cost is critical—redesign may not produce an answer. In this case, the sponsor has to consider giving up some of the planned capability of the system. Otherwise, he has to recognize that the capability cannot be acquired without increasing the cost target. The design process is followed until the program design for a specific software increment has been completed. From that point, development of each increment can proceed concurrently with the program design of the others.

When the development and test of an increment are complete, an estimate to complete the remaining increments is computed. The algorithms used in this computation should reflect the various actual productivity rates experienced in developing and testing previous increments. An alternative plan is prepared and reviewed, as previously described, whenever a cost projection is inconsistent with its cost plan. This may also require changes to a baseline design.

Thus software cost management practices provide a uniform methodology for planning, estimating, measurement, and control. The life-cycle definition provides a structure for the identification of cost-estimating parameters and a standard set of references for the entire development process from proposal through contract performance. The design-to-cost practice describes the management control procedures that balance cost, schedule, and functional capability.

software program management

Practices described thus far are directed at all project participants and department managers. They deal with specific details of design and implementation. They also cover technical and cost-control procedures. One more set of practices is needed to hold the software engineering program together. This final set is directed at the program manager and identifies the project-level plans, controls, and technical management considerations that are necessary for effective software development in a functional organization. Since, in a functional organization, project resources are drawn from several discipline-oriented departments, the program manager does not have direct line authority over all the participants. Program management is much like managing subcontractors in a building construction project. Each subcontractor is highly qualified in a fairly narrow area and is most effective when carrying out a task in that area. The program manager, then, must define tasks clearly, assign them to appropriate departments, define working relationships and technical interfaces between departments, and establish reports and controls to see that the functional groups are carrying out their assignments.

Program management responsibilities include developing and maintaining an organization plan that identifies departments involved, their reporting relationships, and individual department charters. A project work responsibility matrix lists each work task, the responsible manager, and the prime and support roles of the functional groups. The work responsibility matrix should include a dependency network that indicates the predecessor and successor relationships for each task. All cost accounting, status reporting, management accountability, and technical performance are structured around the listed tasks.

From a control viewpoint, program management responsibility includes ensuring system requirement traceability through design and test, providing an architecture control methodology, and managing computer resource loading reserves. These reserves are determined at design time to accommodate design and implementation uncertainties.

Each functional group—hardware engineering, software engineering, product assurance, etc.—must respond to the program manager with a plan of action and a commitment to carry out that plan to fulfill the assigned responsibility.

The software function produces a Software Development Plan (SDP) that describes each assigned task from the work responsibility matrix. The description should cover product schedule, intermediate milestones and schedules, and external dependencies with required scheduled dates. The program manager is responsible for monitoring and controlling the external dependencies specified by the software function. The SDP is updated monthly and incorporates accomplishments, problems, and plans for the subsequent month. It is used as a control document within the software function and as a coordination document in the program manager's office.

Concluding remarks

The software management practices describe the plans and controls for the software engineering environment. These plans and controls reflect the business responsibility of the software function to increase the visibility and understanding of its developing product. Cost management emphasizes cost estimation planning and control. Program management emphasizes effective communication between the software function and the other functions on the project. The technical reviews and design-to-cost practices integrate the application of the design and development practices with the management of the software process. Collectively, these components of software engineering define a structured and predictable approach to managing software projects.

Application of these practices in the Federal Systems Division has improved our ability to predict program behavior. Capability, cost, and schedule still vary from initial program estimates; however, the variances are typically less than experienced in the past. Early identification of deviations from plan has led to timely corrective action. The net result is that the integrated software engineering practices of FSD permit us to deliver high-quality, cost-effective software products with low business risk.

CITED REFERENCES

1. H. D. Mills, "Software development," *IEEE Transactions on Software Engineering* **SE-2,** No. 4, 265–273 (December 1976).
2. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, Inc., New York (1972).
3. C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM* **12,** No. 10, 576–583 (October 1969).
4. R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Inc., Reading, MA (1979).
5. N. Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1976).
6. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1973).
7. A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," *Proceedings of IEEE Comsac '77*, IEEE Catalog No. 77Ch1291-4C, 242–251, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854 (1977).
8. H. D. Mills, *On the development of systems of people and machines*, Springer-Verlag, New York (1975).
9. D. L. Parnas, "The use of precise specifications in the development of software," *Proceedings of IFIP Congress 77, Toronto, August 8–12, 1977*, B. Gilchrest, Editor, North-Holland Publishing Co., New York (1977), pp. 861–867.
10. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).
11. C. A. R. Hoare, "Monitors: An operating system structure concept," *Communications of the ACM* **17,** No. 10, 549–557 (October 1974); "Corrigendum," *Communications of the ACM* **18,** No. 2, 95 (February 1975).
12. N. Wirth, "Toward a discipline of real-time programming," *Communications of the ACM* **20,** No. 8, 577–583 (August 1977).
13. H. D. Mills, "Software engineering," *Science* **195,** No. 4283, 1149–1205 (March 18, 1977).
14. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold Co., New York (1971).
15. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* **11,** No. 1, 56–73 (1972).

16. M. A. Jackson, *Principles of Program Design*, Academic Press, Inc., New York (1975).

17. B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation* **14,** No. 5, 48–59 (May 1973).

18. R. W. Wolverton, "The cost of developing large-scale software," *IEEE Transactions on Computers* **C-23,** No. 6, 615–636 (1974).

19. T. C. Jones, "Measuring programming quality and productivity," *IBM Systems Journal* **17,** No. 1, 39–63 (1978).

20. C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal* **16,** No. 1, 54–73 (1977).

21. R. Yeh, Editor, *Current Trends in Programming Methodology*, Vol. 1, Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).

22. G. J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold Co., New York (1978).

23. R. C. McHenry and C. E. Walston, "Software life-cycle management: Weapons process developer," *IEEE Transactions on Software Engineering* **SE-4,** No. 4, 334–344 (July 1978).

24. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Publishing Co., Inc., Reading, MA (1975).

25. C. L. McGowan and R. C. McHenry, "Software management," *Research Directions in Software Technology*, P. Wegner and W. Wolf, Editors; to be published.

26. E. A. Goldberg; "Applying corporate software development policies," *Software Development: Management, the Seventy-First Infotech State of the Art Conference*, London, May 12–14, 1980, Infotech, Ltd., Maidenhead, England (1980).

27. E. W. Dykstra, "Co-operating sequential processes," *Programming Languages*, Academic Press, Inc., London (1968), pp. 43–112.

28. M. V. Wilkes, "The outer and inner syntax of a programming language," *The Computer Journal* **11,** 260–263 (May–November 1968).

29. B. H. Liskov and S. N. Zilles, "An introduction to formal specifications of data abstractions," *Current Trends in Programming Methodology*, Vol. 1, R. Yeh, Editor, Prentice-Hall, Inc., Englewood Cliffs, NJ (1977), pp. 1–32.

30. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15,** No. 3, 182–211 (1976).

31. J. D. Aron, "Estimating resources for large programming systems," *Software Engineering, Concepts and Techniques*, P. Naur, B. Randell, and J. N. Buxton, Editors, Petrocelli/Charter, New York (1976).

32. M. R. Seldon, *Life Cycle Costing: A Better Method of Government Procurement*, Westview Press, Inc., Boulder, CO (1979).

The author is located at the IBM Federal Systems Division, 18100 Frederick Pike, Gaithersburg, MD 20760.

Reprint Order No. G321-5133.