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

Copyright 1980 by International Business Machines Corporation. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to *republish* other excerpts should be obtained from the Editor.

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

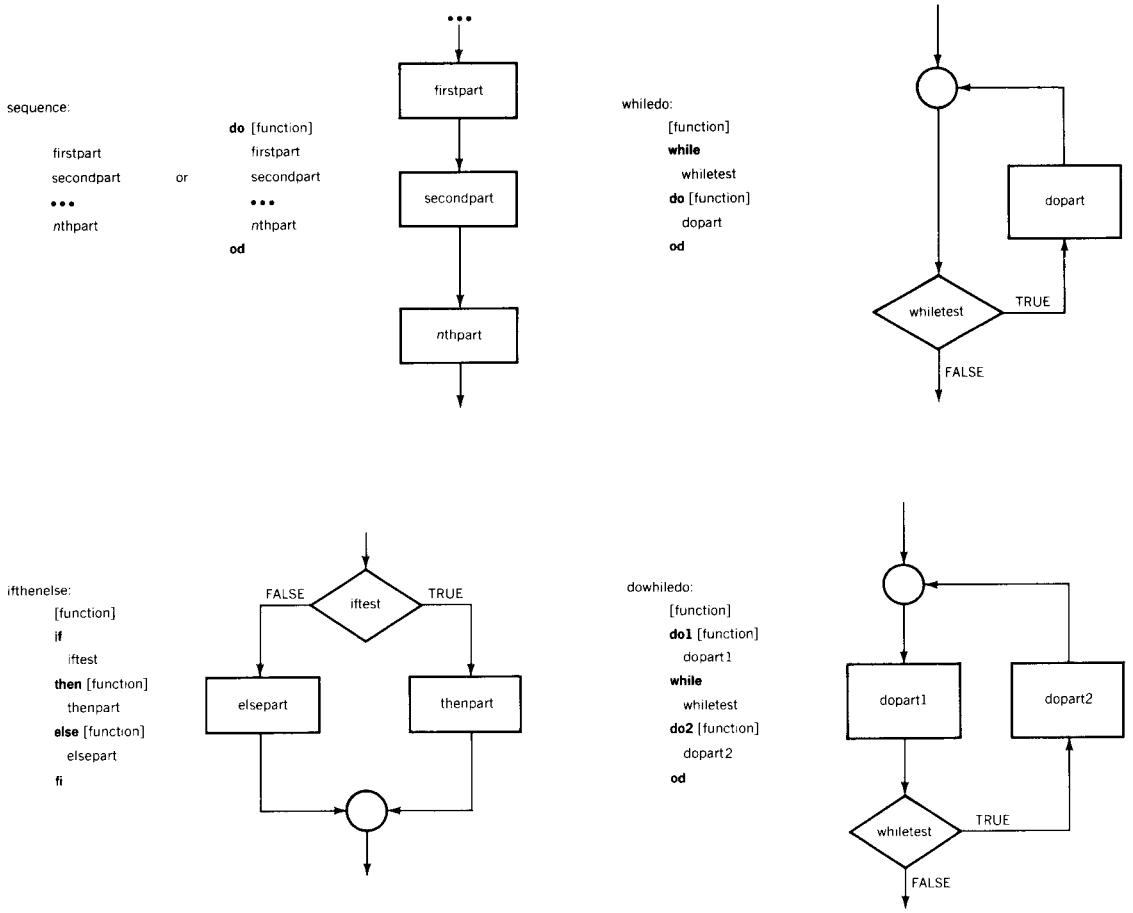
The program expression practice defines requirements for the textual language for recording program designs. This language is intended to support the following activities:

- 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



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 veri-

fication. 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 "$\circ$" 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*.

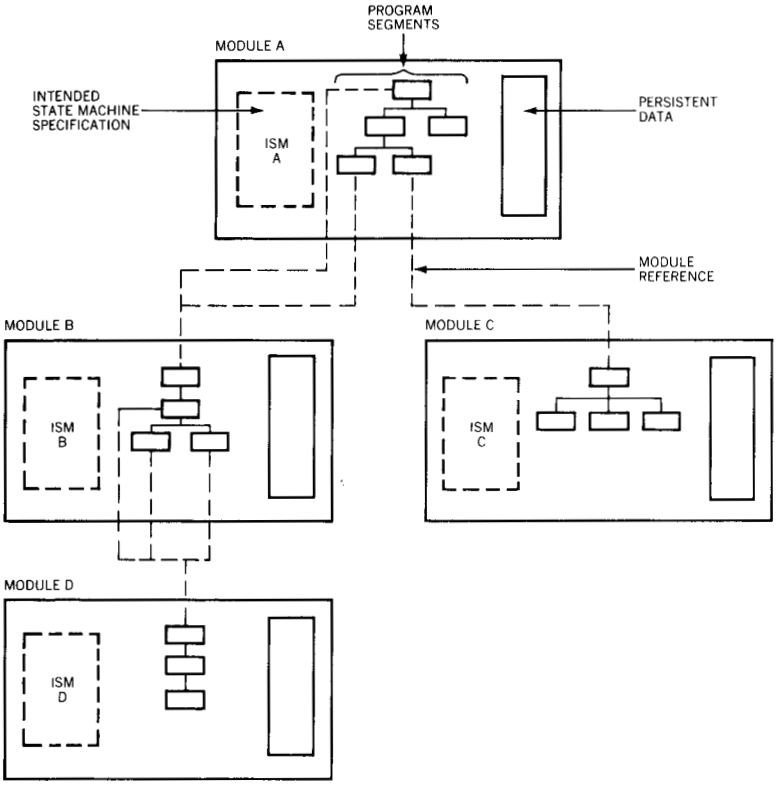
specification by intended state machines

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:

$x_3 = x_2 - y_2$	$y_3 = y_2$
$x_3 = x_1 - (x_1 - y_1)$	$y_3 = x_1 - y_1$
$x_3 = y_1$	$y_3 = x_0 + y_0 - y_0$
$x_3 = y_0$	$y_3 = x_0$

Therefore

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

pass

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