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 mes-

sage 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 develop-

ment 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.

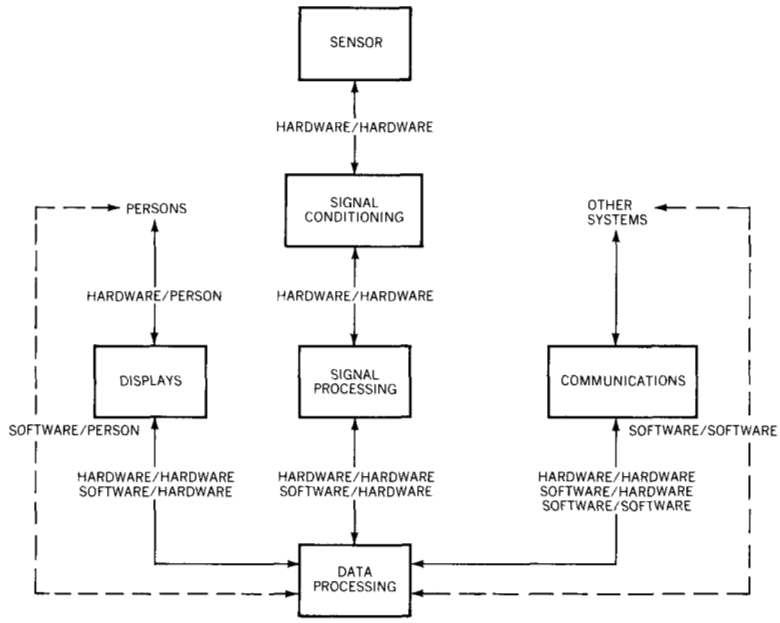
Data recording is a key element of a software system design and is incorporated in a manner that minimizes interference and distortion. The software for each increment is instrumented for measurement of such system resources as prime and secondary storage utilization. The measurements should be performed 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 re-

quirement(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 gov-

ernment 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.