Quality emphasis at IBM's Software Engineering Institute

by M. B. Carpenter
H. K. Hallman

Improvements in quality and productivity in the development of programs can be obtained by instructing the programming development groups in the use of modern software engineering methodology. To provide this instruction for its employees, IBM has established a Software Engineering Institute. Currently training in the methodology is being offered through an education program of the Institute known as the Software Engineering Workshop. This paper describes the role of the Institute, its background and offerings, and some results obtained.

To provide continuing advanced technical education to its technical professional employees, IBM established the Corporate Technical Institutes: the Manufacturing Technology Institute, the Quality Institute, the Software Engineering Institute, and the Systems Research Institute. They offer classes and laboratories addressing critical areas that concern the technical vitality of employees. The Software Engineering Institute and its main programs are the focus of this discussion.

A primary role of the Software Engineering Institute (SEI) is to communicate and facilitate the use of the intellectual foundations necessary to meet the quality and productivity levels mandated by the rapidly expanding and competitive software industry. In order to achieve and maintain these levels, the programmer cannot act as a "skilled craftsman" or as a "high priest" holding power over clients with mysterious knowledge and incantations.[1] Rather, the programmer must be a professional, understanding the disciplines of science and engineering and applying them in a controlled and business-sensitive manner to the development of software products. Such a person is then a software engineer by the definition used throughout this paper.

The evolution of the Institute to its current organizational structure and the educational and administrative methods that characterize it are noteworthy, but equally important is the content of the curriculum chosen. Of course, IBM is not an academic organization. The educational motivation and approach are decidedly different in an academic and in a business concern, although the divergence between the two has narrowed recently. A business is primarily interested in producing a needed, high-quality product in a cost-effective manner, while optimizing the use of available resources. In the software business, the most significant resources are the software developers and the knowledge and skill they possess. The greater the extent to which those resources can be enhanced through better intellectual methods or increased automation of less creative aspects of the task, the bigger the payback, or return.

The Software Engineering Institute is supportive of the business concerns of IBM. Its role is to address

educational and methodological inhibitors to make order-of-magnitude improvements in software quality and its corollary, software productivity. This requires the careful selection of curriculum to provide the maximum quality improvement. The dissemination of the selected content through a large population base such as IBM's programming development groups requires tens of thousands of classroom hours, followed by consultation service back on the job, where the real application of the curriculum occurs. Also required are software development practices to support the development process suggested by the methodology. And of critical importance is the integration of automated development tools into the process to make the application of the methodology both standard and natural for the programmer.

It frequently takes as long as six months for professionals to become comfortable and respected in new jobs.

Organizational evolution

Need. In the late 1970s, it was apparent that some of IBM's more experienced programmers were in danger of having their expertise become obsolete and that many of the computer science school graduates had more relevant if not better technical foundations than our experienced personnel.

An additional phenomenon that had always been a concern, the terminology gap, was creating problems in our ability to absorb these new graduates quickly into our existing software projects. The terminology used by experienced programmers in industry was different from that used in the literature and in educational institutions. Thus, a newly hired computer science graduate had to spend considerable time formally or informally learning the terminology used in the company before becoming fully productive in using skills acquired in college. Experience has shown that it frequently takes as long as six months for professionals changing job assignments to become comfortable and respected in their new jobs. But with computer science graduates entering industry for the first time this period can take as long as two years. Within industry circles in the past, this lengthy adjustment was attributed to the graduate not getting the proper training in college. Today there is a different perception. Recent graduates from schools with good computer science curricula appear to be better prepared technically than ever before. They sometimes have better understanding and technical know-how than those who have been in the profession for many years. However, the terminology gap and the burden of learning a new vocabulary can keep them from being productive.

A little perspective is needed to understand why this problem exists. Programmers in the industrial development laboratories have been asked from the very beginning of the profession to create solutions to problems that have not been solved before. This situation is true in all aspects of the programming profession, but is specifically the case in systems software development. In creating new solutions, one frequently has to develop new terminology to suit the new environment and solution. As the new product is used and the creators go on to other projects which are similar yet different, the new terminology becomes widespread throughout a company's internal programming community. This is good and to be expected and fosters communication across the various software organizations.

In the past, internal standards frequently were created to foster the use of a common terminology, for example, "buffer," "communication area," "save area," "indirect addressing," and "linkage register." Since programmers were not being trained in the universities in the early days of the industry, the computer manufacturing companies trained their own programmers using this new terminology. The problem was that in many industrial environments, time was not allocated to publish these new developments in the software trade publications as they occurred. The terminology sometimes found its way into the reference manuals for the products, but technical descriptions were usually missing. For example, data-driven logic algorithms had become well developed by the mid-1960s but appeared in the code with very few explanations.

In the late 1960s and early 1970s, as more educational institutions were being encouraged to do re-

search in the software sciences, there was a lack of published work from the industrial sector of programming. The scholars began publishing works of their own creation. These were frequently solutions to new problems. But there were also frequently new solutions to old problems with a different terminology. With the industrial programmers not publishing their work, it was not generally known that a problem had already been solved until after the work was published and translated. For example, table-searching algorithms were rediscovered in the universities in the late 1960s.[2] This phenomenon has caused many problems in the software industry. It has also caused the terminology/vocabulary gap mentioned above and has made it difficult for the industrial programmer to read and understand the literature in his profession. As computer science curricula were being developed, it was natural for the terminology in the literature to be used. For example, what industry calls a "buffer" is referred to as a "list" in university circles. Another example is the "linked list," which is often called a "chained control block" in industry.

Direction. The technical currency and vitality of the programming professional community has been studied in IBM for some time. By 1981, it had come to executive attention. Dr. A. Anderson, then IBM Senior Vice President and Group Executive of the Data Processing Product Group, chartered a task force to do a technical assessment and make appropriate recommendations. The task force recommended that a Software Engineering Institute be formed to provide courses in all aspects of the software development process. It recommended that in the near term the institute should concentrate on the technology transfer of the design methodology work of Harlan Mills and his associates[3] into the commercial systems programming departments of IBM. This had already been done in IBM's Federal Systems Division (FSD) through their software engineering program, which is described in the set of papers titled "The management of software engineering."[4]

Action. The task force recommendation was accepted; the FSD courses on design methodology were modified to be applicable to commercial systems programming. The commercial divisions are now engaged in enrolling their programming development and management staff in the Software Engineering Institute program—the Software Engineering Workshop (SEW). The curriculum concentrates on the use of terminology widely published in the literature and teaches software developers a means to ensure that their design is correct before coding and implementation begin. In cases where this methodology has been used, the number of errors introduced into the code has been greatly reduced, and the goal of defect-free code as well as its achievability can be seen. See the later section entitled "Effect on quality" for more details.

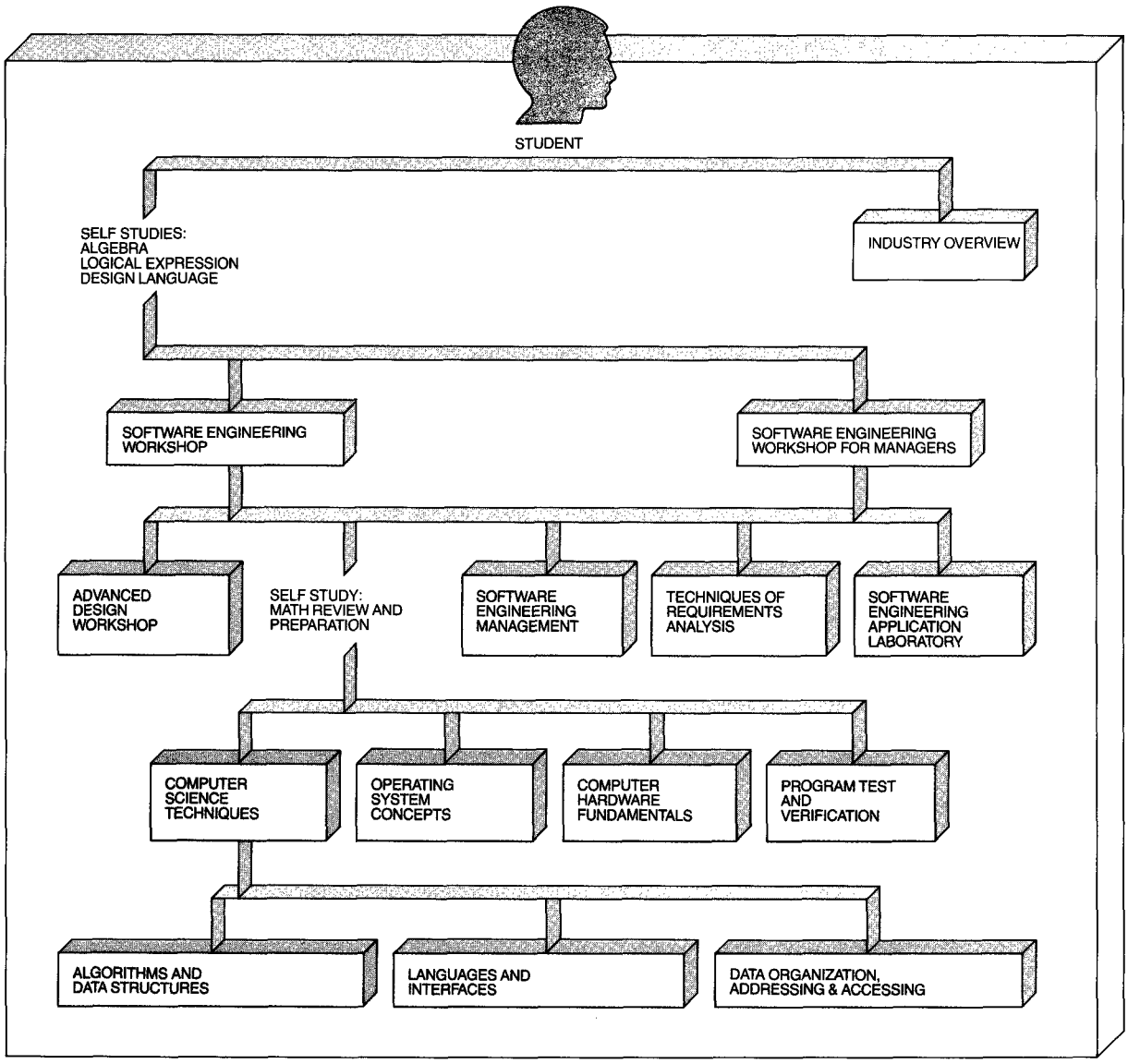
Prior to this decade, the primary means of addressing the goal of zero defects in software products were inspections and testing.

Status. To date, over 45 Workshop instructors have been trained and certified by the Software Engineering Institute, and over 250 classes have been taught. The Workshop courses are being taught to all levels of management and to all the professional personnel in most of the software product development projects within the company. A program of this magnitude has never before been undertaken within IBM. The benefits occur in many ways, not the least of which is the signal to the IBM programming professionals that they are expected to become and remain technically up-to-date.

The curriculum

Prior to this decade, the primary means of addressing the goal of zero defects in software products were inspections and testing, i.e., defect removal. The industry seems to have reached a plateau in the area of defect removal. We have a very finely tuned defect removal process, but wringing further improvements from it is very difficult. In our pursuit of the goal of zero defects, another avenue is available and must be taken: defect prevention. Defect prevention means initially constructing provably correct products rather than unintentionally building in defects and later detecting and removing the faults. Defect prevention techniques have been used very successfully both within and outside of IBM. The central thrust of the Software Engineering Institute's curric-

Figure 1 Software Engineering Institute curriculum



ulum is the infusion of defect prevention techniques within the software development process.

Figure 1 illustrates the present curriculum of the Software Engineering Institute and shows how a student might progress from one course to another. In general, the student should take paths through the curriculum from the top to the bottom of the chart. A description of each course can be found in the *Software Engineering Institute 1985/86 Bulletin.*[5]

Software Engineering Workshop

The course. The foundation course, the Software Engineering Workshop (shown in light shading on Figure 1), is a two-week class focusing on disciplined, precise, and verifiable recording of software design. Its primary audience comprises all system programmers and managers of software development projects within IBM. In addition, there is a one-week version of the Workshop targeted for executive managers

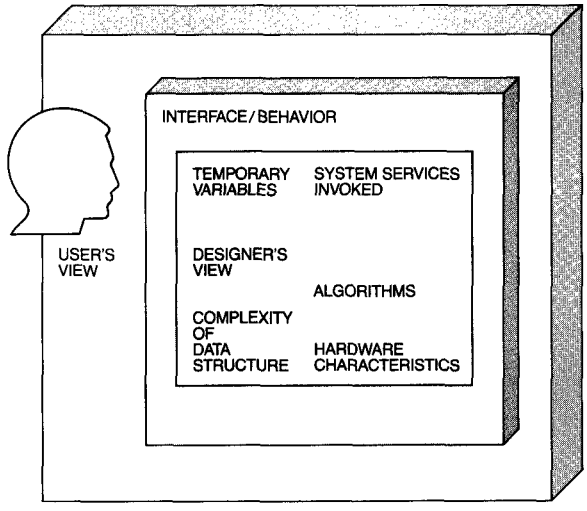
who need an appreciation of the concepts as they affect software products and processes. The Workshop is also presented in a 12-session format as part of the curriculum of the IBM Systems Research Institute, where graduate-level college credit can be earned for the course. The Workshop introduces the use of mathematical models to describe software entities, concepts of abstraction and encapsulation, and design verification.

Why this focus on design recording? Through many industry-wide studies it has been shown that most of the defects in a software product are introduced in the design phase. On average, without defect prevention techniques, 60 defects per thousand lines of code[6] will be injected.[7] Also on average, 42 of them will be injected prior to the coding phase. As software developers, we then spend a great amount of time, effort, and money detecting and removing those errors through such techniques as reviews, inspections, and testing. Some estimates show that half of the development expense goes to some form of defect detection/removal activity. The message of defect prevention is that it is highly cost-effective. However, since design is a human-intensive activity, it is unlikely that we will universally prevent all defects. Short of total prevention, the next best thing is to detect and remove defects earlier in the process, before they manifest themselves in code and documentation errors. Therefore, the focus of the Software Engineering Workshop is on correct design recording: fewer defects injected in the design phase and the ability to detect those errors earlier through peer inspections driven by mathematically based correctness reasoning.

The Workshop is most commonly taught in a two-week format. The first week emphasizes procedural abstraction, using the mathematical function as a conceptual model for operations on data. The second week emphasizes the abstraction of the representation of data (data abstraction), using the "state machine" as a model for user-defined data types. The subject matter of the two weeks is interrelated: The first week, by pedagogical necessity, limits itself to very simple data types; the second week relies upon the functional expressiveness taught in the first week to define allowable operations on objects of a user-defined data type and builds upon the mathematical function model to portray the state machine model. For each week, the material content has a pattern:

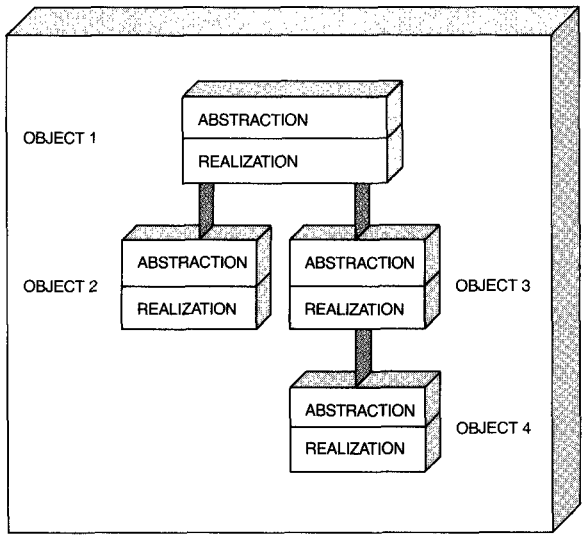
- The concept is introduced that every software object (procedure or data object) can be viewed in two different ways. One view is that seen by the user of the object. The user may be a person, if the object is at the human interface level, or may be another software object, as is more often the case. The user's view should be as simple and sparse as possible, while being both complete and expressive. The other view of the software object is the designer/implementer view. This view shows the actual representation of the object, with the complexity and details (not necessary to the user's understanding of the object) made explicit. The message is that the two views should be kept separate. Figure 2 illustrates the concept of an abstract user's view of a more complex actual representation.

Figure 2 Two views of a software object



Use of the Workshop methodology encourages increased modularity. Increased modularity facilitates the recording of "good" designs by the criteria of good modularization described by Myers:[8] maximized module strength and minimized module coupling. Module strength is characterized by the performance of a single specific function rather than a multiplicity of unrelated functions. Module coupling is a measurement of intermodule relationships and dependencies (often involving knowledge of the internal structure of the module or its data). It is important that a clear and specific user's view be kept independent of the designer's internal view. A methodology that supports a "two-views" philosophy of design presents the opportunity for precise statements of modularity

Figure 3 Hierarchy of abstractions and their realizations



decisions regardless of the criteria of "goodness" espoused by the designer.

- Next the mechanisms for presenting the user's view (writing the specification) are addressed. The underlying mathematical model of the software object, be it procedural logic or data, is presented, and a notation for expressing the specification for the object is taught. In writing specifications, the students are encouraged to balance the precision of mathematical notation with the expressiveness of natural language annotation.

- Following the specification recording, the process of *recording* designs in a top-down manner is illustrated. The process of design *discovery* is hardly ever top-down. It tends to be partly top-down, partly bottom-up, and partly lateral as relationships among other software objects are considered. The process of correct design recording is, however, a top-down process, commonly referred to as "stepwise refinement." (The term "stepwise refinement"[9] was widely published among computer professionals by Niklaus Wirth in 1971.) The philosophy of stepwise refinement is that movement from a higher to a lower level of definition should be taken in small, and therefore verifiable, steps, with each step containing intermediate specifications which become the starting point for lower levels of refinement. Designs so produced are hierarchical networks of interacting objects.

Figure 3 illustrates the notion of hierarchical architectures of abstractions and their realizations. Here object 1 is composed of both an abstraction (an interface and behavior view) and a realization (actual representation or designer's view). The realization of object 1 makes use of two lower-level abstractions: that of object 2 and that of object 3. The use of objects 2 and 3 by object 1 is dependent on their abstract view; object 1 is independent of their realizations. The realization of object 3 makes use of yet another object lower in the hierarchy: object 4. The complexity of the realization of each object is isolated from the user/invoker of that object.

- The subsequent pedagogical step is the definition of the process of determining the correctness of each step of refinement. The process is called "verification." A means of recording "proofs" or correctness arguments is shown. These recorded proofs are not, however, the goal of the instruction in verification. For each construct within the design language, there is a set of mentally applicable questions which the student is encouraged to make a part of his/her habitual practice of design creation. For example, for a looping construct, one of the questions in its verification set would concern loop termination. The emphasis is on the use of the correctness questions to examine each design step prior to introducing it into the software product. Thus, the goal of verification is termed "constructive correctness," a key aspect of the overall theme of defect prevention.

The Workshop is taught by a combination of lectures, classroom exercises, discussions, homework exercises, ungraded quizzes, case studies, and graded tests. Successful completion of the Workshop requires participation in a significant team exercise (case study) and achieving a passing average on the two graded tests, one for each week. The subject matter is introduced in gradually increasing levels of complexity. The class size is targeted at 25 students so that the creation of a constructive, friendly atmosphere between student and instructor is feasible. There are generally two instructors who share the instructional load, which is quite intense, with 80 hours of classroom time during the two-week period. Individual assistance to students is available before and after class. Every effort is made to make the Workshop experience a positive and successful one for the student.

The Software Engineering Workshop, as part of a quality improvement program of a business organi-

zation, produces a somewhat unique educational atmosphere. Little emphasis is placed on graded activities, but the presence of tests is a motivator for learning. The grades serve as a measure of minimal competency in the material and indicate general trends in instructional quality. The only data kept after the class on a student's achievement is a record of successful completion of the course. The grades are not distributed and are not used as a measure of the employee's job performance. The Software Engineering Institute's expectation and experience with this Workshop is that half the students will achieve an average of 90 or above on the two tests and that less than five percent of students will score below 70. The Workshop is not a means of ranking employees or of effecting career changes. It is a success-oriented program for establishing a universal foundation in certain software engineering principles, thus enhancing communication among software professionals and moving the total organization toward the goal of zero-defect software products.

The selection and training of instructors for the Workshop are critical to its success.

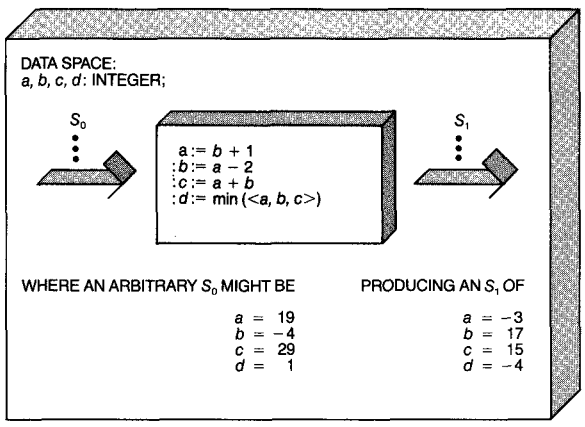
The selection and training of instructors for the Workshop are critical to its success. Instructors are drawn from IBM's own software development community. Qualities primarily sought are extensive experience in software development, a strong mathematics background, and peer leadership. The ability to relate to students and their experiences at work is essential to instructor credibility. An instructor candidate must first successfully complete the Workshop as a student and then go through an intensive certification program. Certification requires full-time concentration for up to a year. The candidate must do in-depth studies of the technical content of the course, prepare and give each of the 21 lectures before an audience of other instructors who determine how ready the presenter is to teach the lecture, and conduct all the lectures plus 11 other review/discussion sessions in live classes. During candidacy, the instructor is expected to practice applying the methodology and using methodology-related tools. Typically, a certified instructor is assigned for two years to an IBM site where software is developed. At each of these "satellite" locations of the Software Engineering Institute, instructor teams will run workshops on the methodology of the Software Engineering Workshop and provide technology consulting service to local projects. At present there are 20 active satellite locations worldwide.

The student materials for the Workshop include a four-volume set of lecture notes.[10] These volumes contain copies of material presented during the lectures, student exercises with answers, and essays or self studies related to individual lectures. One of the four volumes is a reference manual describing the design language used as a vehicle for design expression during the class and on projects after class. Each student receives a textbook, *Structured Programming: Theory and Practice*, by Linger, Mills, and Witt.[3] The classroom contains a selection of books on subjects covered in the Workshop or applicable to other courses in the Software Engineering Institute curriculum.

The Workshop methodology. The methodology taught in the Workshop can be termed a "function-based" methodology. Software may be defined in simplified terms as "operations on data." Those operations are modeled in the Workshop on the mathematical function; i.e., each operation might be viewed as a set of ordered pairs mapping inputs to outputs, where each input value is unique. What is distinctive about the application of the function model in the Workshop is the definition of the set from which the values in the function elements are drawn. Often a function is used to map inputs contained in one set to outputs contained in a different set. In the Workshop the model is used to map input states of all the variables known to the operation to output states of all those variables. The model is divorced from the concept of modes of parameters and applies to functions defined at all levels of design, i.e., functions subordinate to external interfaces. The set of first elements used in the function is called the domain; the set of second elements is called the range. Both the domain and range are within the set of declared values, termed the data space. The function rule records which second element will be produced given a first element.

The notation used to record functions is called the "concurrent assignment statement." It describes the

Figure 4 Mathematical function applied to software operation



simultaneous transformation of the current data states of all variables to their new data states. Figure 4 illustrates the use of a concurrent assignment statement to express a function. In the example, the data space is defined by the variable declarations. The diagram shows the results of a function acting upon a sample data state and producing a new data state. The statement of the function expresses *what* is to occur and suppresses all procedural logic defining *how* the output might be attained. As the operation is viewed as concurrent, there is no concept of intermediate data states. The function is an opaque box view of the operation.

Defining function rules is relatively natural to a programmer, but specifying the domain (set of legal inputs) of a function is not. Using the mathematical function model adds the discipline of explicitly recorded domains. In cases where the domain is not equal to the data space, domains must be clearly understood by both the user and the designer of the function. In the example of Figure 4, the domain is equal to the data space. All of the illustrated operations keep data within the declared bounds because the set of integers is closed under the operations of addition and subtraction.

If we change the data space to

a, c, d :INTEGER
b :INTEGER ≥ 0

the domain is then less than the data space. The function must be guarded to ensure that a negative integer is not assigned to b. To express the domain of the function over the revised data space, one would record

$$\begin{aligned} a \geq 2 \rightarrow \quad & a := b + 1 \\ & : b := a - 2 \\ & : c := a + b \\ & : d := \min(\langle a, b, c \rangle) \end{aligned}$$

thus indicating that there are no ordered pairs in the function which contain an input data state with a value for a less than 2.

Through a process known as "stepwise refinement," a function may be refined into a program whose control structures are members of a predefined set of control structures native to the design language. At each step of refinement, more *how* information is added to the design. Function boxes, viewed as opaque, are expanded into control structures, which in turn contain other function boxes. Figure 5 illustrates this process. Letters represent functions (assignment statements) and predicates (expressions that produce a Boolean value).

A distinguishing feature of the Workshop methodology is the retention of function abstractions in the design recording (as illustrated in Figure 5) as intermediate specifications in the square brackets ([. . .]). They allow the reader to understand *what* the more detailed design is accomplishing without the underlying complexity of the design. Furthermore, intermediate specifications form the basis for reasoning about the correctness of the design.

Since every assignment statement reflects a mathematical function, correctness reasoning (called verification) involves applying mathematical concepts to the functions. The verification technique for a sequence control structure consists in first deriving the mathematical function of the sequence using the concept of function composition and then comparing the result to the specification. Alternative branching programs are verified by partitioning the domain of the specification by the program predicate(s) and comparing the functions in the specification and in the program in corresponding partitions. Looping programs are verified by first ensuring loop termination through arguments based upon finite sets. When it has been determined that the loop will produce a final mapping, a function-based verification technique, making use of the iteration recursion theorem as covered in the text,[3] is taught as an alternative to methods based on loop invariants. The verification techniques are easy to apply and present the potential for certification of the correctness of a

design. Essential to the simplicity of the verification process is its application to relatively small pieces of the design, each with a precise specification. The process can be applied mentally, verbally, or in written form at each step of the refinement process, as shown in Figure 5.

A corresponding stepwise refinement process can be applied to data. The mathematical model for data is the state machine, where operations can be viewed as functions applied to the data modeled. Figure 6 depicts the state machine model and its mathematical function view.

Each operation uses as input any external data values (i) and the current value of the data modeled (c). It potentially produces external data values (o) and a new value of the retained data (n). Thus viewed as a function, each operation is defined by a set of ordered pairs, composed of $((i,c), (o,n))$.

The state machine model is commonly applied to a particular instance of data. The Workshop extends the model to be applicable to a data type, i.e., a family of data instantiations with common characteristics. Furthermore, there may be two different views of each data type. One view presents the simplest possible structure in which to describe the information content of the data type and in which to define the set of allowable operations upon data of this type. Abstract structures are often defined in terms of sets, lists, or maps. Each operation is clearly defined in terms of its interface and behavior, again described in the format of a concurrent assignment statement. The other view, the designer's, describes the actual data representation structure. The operations are restated in terms of the more complex structure and often have a larger number of partitions of behavioral specification.

Stepwise refinement as applied to data, as with function, is the movement from the abstract view to the actual representation view. The total data transformation may occur in one step or may occur in several steps, where intermediate data abstractions may be used to encapsulate logical partitions of the data entity. As with function refinement, a verification technique is taught to ensure that the transformation at each step is correct, i.e., satisfies the intent of the abstract view. After the operations allowed for the data type are expressed against the actual data representation, they are refined into procedural logic using the same techniques as used for other functions.

Figure 5 Stepwise refinement of function

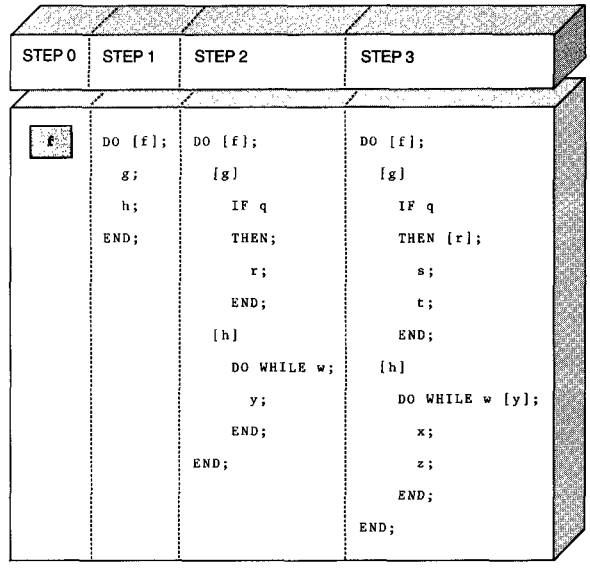
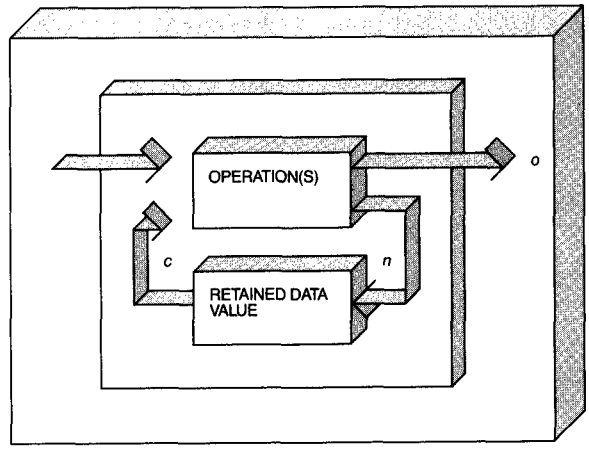


Figure 6 State machine model applied to data



The design product using the Workshop methodology is a network of interacting functions and data objects, each with two distinct views (abstract and concrete). Systems thus developed tend to be easier to understand, more modular, more maintainable, and more free from defects. More precision in the design product eases the task of early prototyping and performance modeling. Since design details are encapsulated (decoupled from their use), alternative representations can be substituted with minimal ef-

fect to the system. As libraries of alternative design representations are developed, the potential for reusability grows, having a positive effect on both productivity and quality.

A significant part of the curriculum is developed and taught by university faculty.

Courses for Software Engineering Workshop graduates. Follow-on courses to the Software Engineering Workshop (shown in medium shading on Figure 1) are designed to apply and augment its concepts.

- Advanced Design Workshop expands the methodology to the concerns of designing mechanisms for multitasking environments.
- The Software Engineering Management curriculum offers to managers of software development projects perspectives on business and managerial issues as they are affected by evolving technologies and processes.
- Techniques of Requirements Analysis extends the discipline of rigorous specification and design to the requirements definition phase of development.
- Software Engineering Application Laboratory emphasizes the practical concerns of applying software engineering methodology to designing in real-world environments and using methodology-specific tools.

University programs. A significant part of the Software Engineering Institute curriculum (shown in dark shading on Figure 1) is developed and taught by university faculty from a variety of cooperating colleges and universities. These courses are elective for graduates of the Software Engineering Workshop. Topics are more typical of those found in computer science curricula and are taught in the style of a university course. Courses in the university programs are typically five days in length.

The Corporate Technical Institutes encourage the sharing of ideas and teaching responsibilities between industry and academia. The cross-pollination is mutually beneficial to the faculties and to the students.

All courses in the Software Engineering Institute's university programs require the Software Engineering Workshop as a prerequisite as well as the completion of a self-study in mathematics. There is a subset of university program courses with only these two prerequisites:

- Computer Science Techniques presents subjects normally encountered by a student in computer science during the freshman and sophomore years. These include graph theory, efficiency measures and notations, and logic design verification using pre- and post-conditions.
- Operating System Concepts examines system design concepts as applied to operating systems.
- Computer Hardware Fundamentals explores the principles of computer hardware design.
- Program Test and Verification focuses on the principles that underlie an engineered approach to the testing phase of software development.

The remaining courses of the university programs additionally require the successful completion of Computer Science Techniques for attendance:

- Algorithms and Data Structures presents methods for analyzing both algorithms and data structures in light of efficiency considerations.
- Languages and Interfaces examines the relationships among tasks, people, and computers as they are communicated through languages and interfaces.
- Data Organization, Addressing, and Accessing focuses on the role of data in software systems, highlighting means of data access and control.

Industry overview. Professionals with responsibility for making high-level decisions concerning products and the marketplace require a broad awareness of the computer industry. Industry Overview is a one-week course which surveys computer technology and products, both hardware and software. The course is taught by university personnel and independent consultants involved with research in the computer industry. The course concludes with an overview of IBM's business planning process.

Past indicators of success and future direction

To date the Software Engineering Institute has had approximately 5000 students worldwide go through

its Software Engineering Workshop and hundreds more through its various follow-on courses. As our motivation is not solely an academic one, the true measure of our success is not classroom statistics but the business impact of the use of the methodology taught.

The first conference to evaluate that influence was the Workshop on Applications of Software Engineering Technology. Held October 16–19, 1984, in La Gaude, France, it featured speakers from 15 IBM sites worldwide. The presentations at this conference illustrated that the curriculum content of the Software Engineering Institute is being applied in a wide variety of environments. Future conferences of this nature are planned.

Further documentation of the methodology usage is distributed internally by the Institute to graduates of the Software Engineering Workshop. The mailing includes copies of papers written by employees about projects in which they applied concepts of software engineering to advantage. The intent of this internal publication is to foster communication among practitioners of the methodology and to encourage people to write about their work.

Effect on quality. The use of the Workshop methodology can have a very significant effect on the quality of the product being developed. The term "quality" in this application is used to refer to intrinsic quality or quality in the form of measurable entities such as defects found per unit of work. It is not necessary to discuss the pros and cons of this type of measurement here; that has been done adequately elsewhere in the literature.[11–13] When one is making the same measurement with the new methodology as was made with the old methodology, the results are indicative of what can be expected from the use of the new methodology.

First let us look at the results on a large systems software type of product. Project A was a large ongoing effort over many years with many versions. It consisted of an operating system, real-time control, data management plus specific application-type code. It existed in an environment where high quality has always been required. Over a period of several years and several versions, it had a historical average of 60 defects injected per thousand shipped lines of code, which is not an unusual number. It is about the norm for the technologies used in the 1970s. A new version of the product was created using the Workshop methodology. It contained significant modifications to the existing system plus large amounts of new code (greater than one hundred thousand lines of code). For areas using the methodology as applied to both functions and data abstractions, the number of defects injected was reduced by a factor of 10. For areas using the methodology but applying it only to the area of functions, the defects injected were reduced by a factor of 3. The techniques for defect detection and removal and their associated yields remained unchanged for the new version. The intrinsic quality of the delivered product was improved by a factor of 12.

Project B was a financial application developed for use within IBM at several locations. Similar applications developed by the same organization had a history of problems during installation and afterward. This was a medium-sized product, about 20 000 lines of PL/I code, developed using the Workshop methodology. No errors were found after unit test. At no time after design was complete were any design defects found. The programmers on this project have become well known among their user community for developing zero-defect code.

The methodology has many subjective effects on quality. The users of the methodology claim that having the design of the product documented in an accurate hierarchical manner allows the user to understand the potential of the product earlier in its life cycle. This knowledge reduces late changes to requirements.

Effect on productivity. Productivity reflects the cost of a product. To get an improvement in productivity, one has to reduce this cost. There are many parts of the development cycle that can be worked on to affect product cost. The industry is just coming to understand the effects of quality on productivity. The two are usually tied together; i.e., if you concentrate on those things that affect the quality of the product, you will minimize the cost of the product.

The Workshop methodology concentrates on preventing errors from getting into the design and thus into the product. This aspect not only improves the quality but also avoids expensive repair actions after the product is shipped. Thus, the total cost of the product is reduced. It also has a very positive effect on the customers' costs.

The part of the development cycle prior to shipment to the customer is, for product development groups, the part of the cycle usually analyzed for productivity

effects. The methodology increases the effort prior to writing of code. The design cycle can take twice as long, but all projects report that coding and testing go very quickly. The experience of Project B was that total time was reduced by 10 percent, even when accounting for the learning curve problems. Design time increased to 70 percent of the total time with the use of the Workshop methodology, up from previous project measurements in which design used 20 percent of the total time. Coding time decreased from 40 percent to 20 percent; and testing decreased from 40 percent to 10 percent. In summary, it is expected that the methodology will not increase the development cycle but will most likely decrease its costs 10 percent or more when the people are over the learning curve with the methodology. Since the methodology concentrates on simplifying the design, greater productivity can be gained where more complex designs are being created.

Risk management is the primary activity that occurs in planning and managing a software project.

The phenomenon of lengthening the design cycle can be very troublesome to management who have been measuring progress on the basis of lines of code. For this reason, the Software Engineering Institute has the strong conviction that a management team should go through the Workshop classes ahead of or with their programming staffs.

Another interesting aspect that affects productivity has to do with how fast new people can be brought into a project and made productive. Project C was adding a component of 20 000 lines of code to an existing operating system. Using the Workshop methodology, they were able to complete the design and coding phases in one sixth of the time it took a sister project to complete. This was done with a similar number of experienced programmers but using two relatively new programmers for each experienced programmer. The new programmers became productive quickly after both the new and experienced programmers attended the Workshop class. The terminology used in documenting the design was that taught in the class. Thus, the terminology gap was removed by having all people brought to the same level. The manager of the project was pleasantly surprised by how quickly the new people became productive.

Effect on risk management. Risk management is the primary activity that occurs in planning and managing a software project. It involves the assurance that there is enough time to fix a problem after it is discovered. The major advances in software engineering in the 1970s had to do with risk management: using the inspection process to find errors earlier in the development cycle so that there is time to fix the problem before shipping the product to the customer.

The Workshop methodology provides a significant addition to the manager's tools for risk management. The manager can actually see the progress in the design development, including the presence of a more formal way to validate the correctness of the design, before coding resources are committed. The testing people have a better organized design to test, with test points already defined. If requirements change, as they frequently do in today's environment, the manager has the design decisions encapsulated so that the impact of the change is isolated and therefore affects a smaller part of the system than it normally would. The implementation of details in the design can be changed and perfected without affecting the user of that portion of the system. The manager has better control over the function in the system, leaving subordinates free to perfect the way to carry out that function. The manager also can have the user community examine an early functional (but not optimized) version of the product before committing it to more detailed design and code. The manager also gets better utilization of the programmers' skills. Most of their time will be spent in creating solutions rather than in finding and fixing defects as in the past. In summary, a manager gets better control of the product development cycle.

Conclusion

The benefits realized by use of a modern software engineering methodology are real. Significant quality improvements are being realized. The methodology has the side effect of increasing the modularity of

software systems, which offers greater ease of design understanding, module robustness, and product maintainability. Underlying all the statistics and all the words of testimony is an important by-product of the Software Engineering Institute's technical contribution to the business: the technical vitality of IBM's employees and their increased awareness of the professional responsibility of continuing education.

Acknowledgments

We are indebted to the Software Engineering Institute staff and instructors worldwide for their contributions to the technical and informational content of this paper. Special recognition is given to those pioneers who have taken what they have learned at the Software Engineering Institute, applied it on the job, and led the way for others. Without the dedication of such people, there would be nothing to write about. Special thanks is extended to Alfred M. Pietrasanta, Bernard A. Rackmales, and Seward E. (Ed) Smith for their careful and considered critique of this paper.

Cited references and note

1. C. A. R. Hoare, "Programming: Sorcery or Science?" *IEEE Software*, 5–16 (April 1984).
2. A. T. Berztiss, *Data Structures—Theory and Practice*, Academic Press, Inc., New York (1971).
3. R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).
4. "The management of software engineering," H. D. Mills, "Part I: Principles of software engineering," D. O'Neill, "Part II: Software engineering program," R. C. Linger, "Part III: Software design practices," M. Dyer, "Part IV: Software development practices," R. E. Quinnan, "Part V: Software engineering management," *IBM Systems Journal* **19**, No. 4, 414–477 (1980).
5. *Software Engineering Institute 1985/86 Bulletin*, G320-6353, IBM Corporation; available through IBM branch offices.
6. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
7. Defects injected are measured by counting all errors found in the product beginning with the point at which inspections start and continuing through the life of the product (or version of the product). Counting usually stops when the product is replaced by a new product or version.
8. G. J. Myers, *Composite/Structured Design*, Van Nostrand-Reinhold Co., New York (1978).
9. N. Wirth, "Program development by stepwise refinement," *Communications of the ACM* **14**, No. 4, 221–227 (1971).
10. *Software Engineering Workshop (SEW) Student Notebook*, Volumes 1–4, Software Engineering Institute, G325-0010, IBM Corporation; available through IBM branch offices.
11. A. J. Albrecht, "Measuring application development productivity," *Proceedings of the Application Development Symposium*, Monterey, CA, Guide/Share (October 1979), pp. 83–92.
12. C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal* **16**, No. 1, 54–73 (1977).
13. K. Christensen, G. P. Fitsos, and C. P. Smith, "A perspective on software science," *IBM Systems Journal* **20**, No. 4, 372–387 (1981).

General references

J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).

B. Beizer, *Software Testing Techniques*, Van Nostrand-Reinhold Co., New York (1983).

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London (1972).

H. D. Mills, *Software Productivity*, Little, Brown, and Co., Boston (1983).

N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1976).

W. A. Wulf, M. Shaw, P. N. Hilfinger, and L. Flon, *Fundamental Structures of Computer Science*, Addison-Wesley Publishing Co., Reading, MA (1981).

Maribeth B. Carpenter *IBM Corporate Technical Institutes, 500 Columbus Avenue, Thornwood, New York 10594.* Ms. Carpenter received her B.A. degree from Duke University in 1966. She then joined IBM, working for the Federal Systems Division (FSD) for 15 years as a programmer, designer, and manager of software for both Air Force and Navy systems. She spent two of those years as an instructor in the Software Engineering Education program of FSD, helping to educate 2300 programmers in the software methodology required by division practices. In 1981, Ms. Carpenter became a faculty member of IBM's Software Engineering Institute, where she is a course developer, instructor, methodology consultant, and instructor trainer.

Harvey K. Hallman *IBM Corporate Technical Institutes, 500 Columbus Avenue, Thornwood, New York 10594.* Mr. Hallman began his career as a programmer in 1956 while in the Air Force, joining IBM in 1960 with a B.S. degree in biochemistry from The Pennsylvania State University. He continued studies for a master's degree in industrial administration from Union College. Mr. Hallman has worked on many diversified projects within IBM, including engineering design automation, MVT for OS/360 in the Model 91, performance-related advanced technology, the Virtual Telecommunications Access Method, finance and insurance industry software support, microcode for the 3695 check processing machine, and process advanced technology. He has been a manager of most parts of the software development cycle: project planning, project control, design, code and unit test, build and test, performance, and advanced technology organizations. Mr. Hallman is currently the manager of Software Engineering Disciplines at the IBM Software Engineering Institute.

Reprint Order No. G321-5243.