BRUCE I. BLUM and THOMAS P. SLEIGHT

AN OVERVIEW OF SOFTWARE ENGINEERING

Computer software has become an important component of our defense systems and our everyday lives, but software development is both difficult and costly. This article examines the similarities and differences between software and hardware development, the essence of software, modern practices used to support the software process, and the application of government methods. We also consider the role of standards and technology centers, and conclude with a view into the future.

INTRODUCTION

Software is a part of everyday life at work and at home. Many things we take for granted are software dependent: watches, telephone switches, air-conditioning/heating thermostats, airline reservations, systems that defend our country, financial spreadsheets. The discipline of managing the development and lifetime evolution of this software is called software engineering.

Software costs in the United States totaled about $70 billion in 1985, of which $11 billion was spent by the Department of Defense.[1] Worldwide, spending was about twice that amount—$140 billion. At a growth rate of 12% per year, the United States will spend almost $0.5 trillion annually on software by the turn of the century.

Studies in the early 1970s projected that software would rapidly become the dominant component in computer systems costs (Fig. 1). The cost of computing hardware over the last few years has fallen dramatically on a per-unit performance basis. That decrease resulted primarily from the mass production of denser integrated circuits. Software remains labor intensive, and no comparable breakthrough has occurred. Thus, the small increases in software productivity have not overcome the increased cost of human resources.

There is broad agreement on what is to be avoided but a diversity of opinions regarding the best way to develop and maintain software. We will examine here why software development is so difficult, what methods are currently available to guide the process, how government methods have responded to those difficulties, and what roads to improvement are being explored. This article, oriented to a technical audience with minimal background in software development, presents a survey of many different methods and tools, for that is the nature of the state of the art in software engineering.

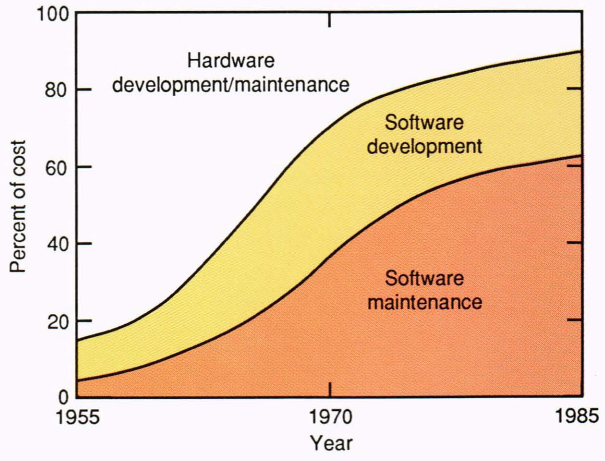


Figure 1-Hardware-software cost trends.

THE ESSENCE OF SOFTWARE DEVELOPMENT

The software process, sometimes called the software life cycle, includes all activities related to the life of a software product, from the time of initial concept until final retirement. Because the software product is generally part of some larger system that includes hardware, people, and operating procedures, the software process is a subset of system engineering.

There are two dimensions to the software process. The first concerns the activities required to produce a product that reliably meets intended needs. The major considerations are what the software product is to do and how it should be implemented. The second dimension addresses the management issues of schedule status, cost, and the quality of the software deliverables.

In a large system development effort, we commonly find the same management tools for both the hardware and software components. These typically are organized as a sequence of steps and are displayed in a "waterfall" diagram. Each step must be complete and verified or validated before the next step can begin; feedback loops to earlier steps are included. A typical sequence is shown in Fig. 2 for software development. The steps are derived from the hardware development model. In fact, only two labels have been changed to reflect the differences in the product under development: software coding and debugging is similar to hardware fabrication, and software module testing is similar to hardware component testing.

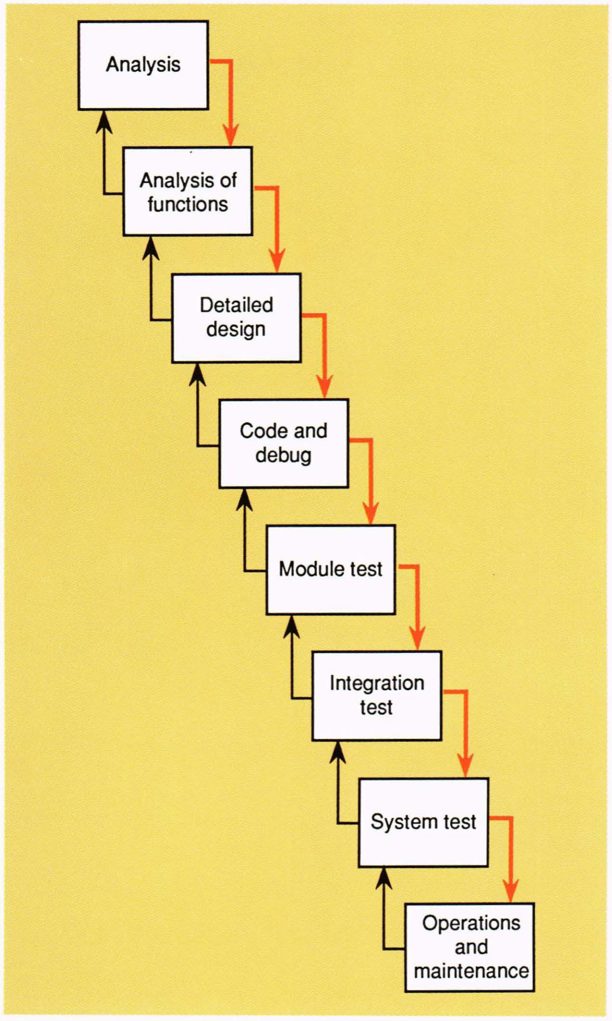


Figure 2— Typical software development steps.

This structural similarity in the flow facilitates the coordination and management of hardware and software activities. There are, however, major differences between hardware and software:

1. Hardware engineering has a long history, with physical models that provide a foundation for decision making and handbooks that offer guidance. But software engineering is new; as its name implies, it relies on "soft" models of reality.
2. Hardware normally deals with multiple copies. Thus, the effort to control design decisions and associated documentation can be prorated over the many copies produced. In fact, it is common to reengineer a prototype to include design corrections and reduce manufacturing (i.e., replication) costs. Conversely, software entails negligible reproduction cost; what is delivered is the final evolution of the prototype.
3. Production hardware is expensive to modify. There is, consequently, a major incentive to prove the design before production begins. But software is simply text; it is very easy to change the physical media. (Naturally, the verification of a change is a complex process. Its cost is directly proportional to the number of design decisions already made.)
4. Hardware reliability is a measure of how the parts wear out. Software does not wear out; its reliability provides an estimate of the number of undetected errors.

These differences suggest that, even with a strong parallel between hardware and software, overcommitment to a hardware model may prove detrimental to the software process. Some common errors are:

1. Premature formalization of the specification. Because the design activities cannot begin until the analysis is performed and the specification is complete, there often is a tendency to produce a complete specification before the product needs are understood fully. This frequently results in an invalid system. Unlike hardware, software can be incrementally developed very effectively. When a product is broken down (decomposed) into many small components, with deliveries every few months, the designer can build upon earlier experience, and the final product has fewer errors. Another development approach is to use prototypes as one uses breadboard models to test concepts and build understanding. Of course, only the essence of the prototype is preserved in the specification; its code is discarded.
2. Excessive documentation or control. Software development is a problem-solving activity, and documentation serves many purposes. It establishes a formal mechanism for structuring a solution, communicates the current design decisions, and provides an audit trail for the maintenance process. But documentation demands often go beyond pragmatic needs. The result is a transfer of activity from problem-solving to compliance with external standards, which is counterproductive.
3. The alteration of software requirements to accommodate hardware limitations. Since software is relatively easy to change, there is the perception that deficiencies in hardware can be compensated for by changes to the software. From a systems engineering perspective, this strategy obviously is inappropriate. Although it may be the only reasonable alternative, it clearly represents an undesirable design approach.
4. Emphasis on physical products such as program code. Because code frequently is viewed as a product, there is a tendency to place considerable store in it. The most difficult part of software design, however, is the determination of what the code is to implement. In fact, production of the code and its debugging typically take one-half the time of its design. Also, most errors are errors in design and not in writing code. Therefore, managers should not be too concerned with the amount of code produced if the design team has a firm understanding of how they intend to solve the problem. And programmers should not be encouraged

to code before they have worked out the full design of their target application. (The exception is the prototype, which is discarded after its lessons have been assimilated into the design.)

If we examine the essential steps of the software process, we see that software development is based on the same general model as that used to build a bridge, conduct a scientific experiment, or manage the development of hardware:

1. First we determine what is to be done (i.e., analysis).
2. Next, we determine how to realize the desired behavior. This is called design, and it includes the allocation of functions, detailed design, and coding. Often this is decomposed into "programming in the large," which involves the architecture of the software system, and "programming in the small," which involves the creation of the code.
3. Following this, we test the product at various levels of system completeness (units, modules, integrated components, and, finally, the full system).
4. Finally, we use the software product, which often changes the environment it was intended to support, thereby altering its initial specification. Consequently, the software will evolve continuously until its structure degrades to the point where it is less expensive to retire it than to modify it. We can view this "maintenance" activity as an iteration of the preceding steps.

Clearly, we can structure these four steps in a waterfall organization. But since true system needs often are not understood without some preliminary experimentation, we also use other development models wherein software evolves from experience with prototypes and earlier system versions. Boehm's spiral model is one example of this revised flow;[2] most process models, however, are built from the four basic activities presented above.

One advantage of the waterfall representation is its long history of use, which has yielded insightful empirical data. For example, a major portion of the software cost is expended on a product after it has been installed. This is called evolution or maintenance, and it can represent one-half to three-quarters of the total life cycle cost. Forty percent of the development cost is spent on analysis and design, 20% on coding, and 40% on integrating and testing (the "40–20–40 rule"). The writing of program code is a very small part of the total cost. A distribution of expenditures for one set of data is shown in Fig. 3. (See the boxed insert for other observations based on empirical data.)

Our discussion thus far suggests that a good approach to software development is one that:

1. Identifies and responds to errors as early as possible in the development cycle.
2. Assumes that there will be continuous change in the product and anticipates its eventual evolution.
3. Minimizes the importance of code production.
4. Maximizes the importance of people, both by bringing experienced people to the difficult tasks and by building the skills of those with less experience.

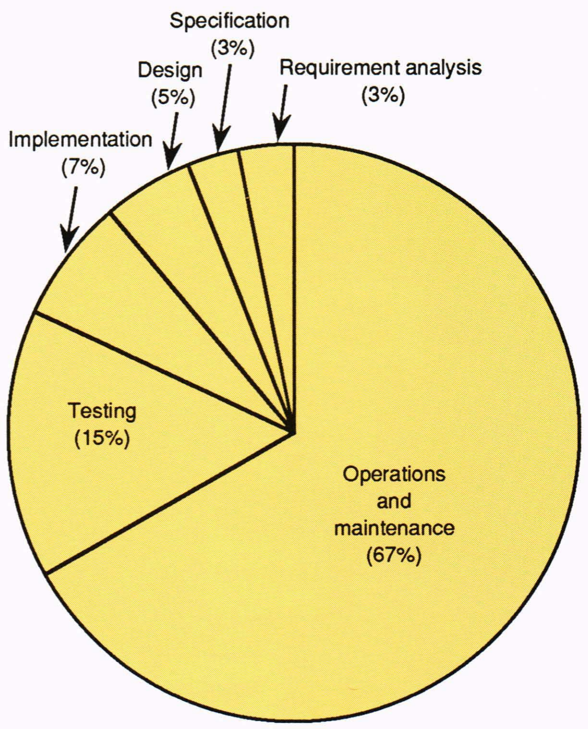


Figure 3— Distribution by cost percentage of the software life cycle. (Zelkowitz, Shaw, and Gannon, *Principles of Software Engineering and Design,* © 1979, p. 9. Reprinted by permission of Prentice Hall, Inc., Englewood Cliffs, N.J.; adapted version appeared in *Computer*, 1984.)

MODERN PRACTICES AND THE SOFTWARE PROCESS

Given our description of the software process, we now address the modern practices used to support that process. We organize the discussion of methods, tools, and environments according to their application to the major process activities of analysis, design (programming in the large), code (programming in the small), validation and verification, and management. (We make no attempt to provide citations for all the tools and methods described. References can be found in the most modern software engineering textbooks.[4])

Analysis

The objective of the analysis is to produce a description of what the software system is to do. Naturally, this will depend on the domain of application. For example, in an embedded application, the system engineers may have specified all the software requirements as part of the system decomposition process; the functions, timing constraints, and interfaces may already be prescribed, and the design can proceed. But as often happens with an information system, the initial intent may be stated only vaguely, and an analysis of the existing operation, along with a study of how automation may help, will follow. The result will be a specification of the product to be implemented.

Although the parameters affecting costs, scheduling, and quality vary from project to project, there are some generally accepted trends. Boehm recently identified the following ten most important industrial software measures or metrics:[3]

1. Finding and fixing a software problem after delivery can be up to 100 times more expensive than finding and fixing it during the phases when the requirements and early design are determined.
2. You can compress a software development schedule up to 25% of nominal, but no more.
3. For every dollar you spend on software development, you will spend two dollars on software maintenance. (Other studies have shown that the costs associated with perfecting the product represent the largest maintenance category, and costs associated with making corrections represent the smallest category. The remaining resources are used to adapt the software to altered requirements. Figure 4 illustrates a typical allocation of costs among maintenance categories.)
4. Software development and maintenance costs are primarily a function of the number of source instructions in the product.
5. Variations between people account for the biggest differences in software productivity.
6. The overall ratio of computer software costs to hardware costs has gone from 15:85 in 1955 to 85:15 in 1985, and this trend is still growing.
7. Only about 15% of a software product development effort is devoted to programming.
8. Software systems and software products each typically cost three times as much per instruction to develop fully as does an individual software program. Software system products cost nine times as much.
9. Walk-throughs can catch 60% of the errors.
10. Many software phenomena follow a Pareto distribution: 80% of the contribution comes from 20% of the contributors.

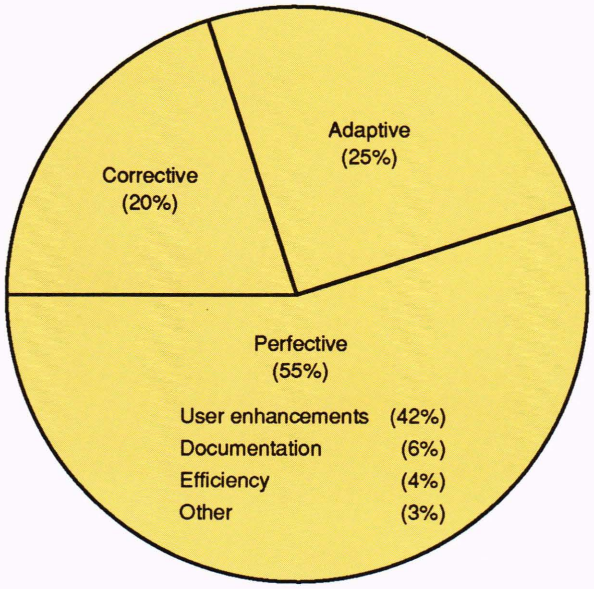


Figure 4— Allocation of system and programming resources to three maintenance categories (reprinted by permission, "Software Engineering: Problems and Perspectives," *IEEE Computer,* © 1984, IEEE).

A common method of analysis is called "structured analysis." The operational environment is first modeled as a network of input–output transformations and documented in the form of "data flow diagrams" (DFDs). The nodes in the DFDs represent the transformations, and the arcs represent the data flowing to and from the transformations. Each node is given a title suggesting the activity the transformation represents, and each arc is given the title of the data in the flow. To convey meaning, abstraction is used to reduce the level of detail. For increased information content, each node can be expanded as a DFD; the only restriction is that all data flows to and from that node are retained as inputs to and outputs from the lower-level DFD.

With this approach, one typically models the physical environment and then draws a boundary separating the automated system from the nonautomated system. Data flows crossing that boundary represent the application interfaces. Next, the functions within the boundary are reorganized to provide a more effective implementation of what previously was a nonautomated process. All flows (arcs) and actions (nodes) are labeled, and the nodes are further decomposed into DFDs until each node is well understood. Because the arcs represent abstractions of the data in the flow, "data dictionaries" are created that detail the data organization and content. There are several variations of structured analysis. In the method developed by DeMarco and Yourdon, the lowest-level nodes are described in process- or minispecs that use "structured English" to detail the processing that the transformation must conduct. A sample DFD, dictionary, and minispec are shown in the boxed insert.

Most structured analysis techniques are designed for information processing applications. The initial goal of this method was to provide a manual technique that would allow the analyst to detail his thoughts systematically and communicate the results to the sponsors and users. Recently, automated tools have been developed to assist in drawing the DFD and maintain dictionaries of the transformations and flows. (Such tools are known as CASE: computer-assisted software engineering.) Variations of the DFD also have been adopted for use with real-time systems by adding symbols to model queues and messages transmitted among nodes.

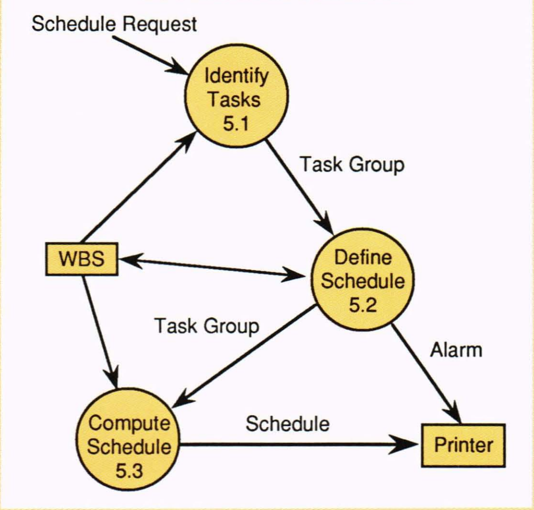
The requirements analysis is conducted in a top-down mode. This decomposition approach imposes some design decisions on the product; for example, the DFD essentially establishes the module structure for the implementation. Some suggest that this is a weakness of such methods: the analyst must make critical design decisions when he least understands the problem being solved. The alternative is a "composition" approach in which one models portions of the system that are well understood and builds the system from those components.

STRUCTURED ANALYSIS DESCRIPTION

The figure below contains a simple example of the representation used during structured analysis. For this data flow diagram (DFD), we assume that there is a parent DFD, with at least five bubbles or activities. This diagram is an expansion of the bubble, 5.0, Determine Schedule, of the parent activity.

Typically a DFD contains five to nine bubbles, although only three are shown. Each bubble is labeled with the activity it represents; the data flows to and from each bubble are labeled; and the data stores (i.e., the file containing the work-breakdown-structure [WBS] data) and external elements (i.e., the printer) are identified with their special symbols.

5.0 DETERMINE SCHEDULE



Because the processing for this DFD is clear, there is no need to expand it to another DFD level. Bubble 5.2, Define Schedule, is described in a minispec, which conveys the processing while avoiding the detail required of a programming language. For example, "get and list WBS# and WBS TITLE" is several instructions, and the reenter statement after printing the error message implies a GOTO (not shown).

Finally, the data dictionary defines the major elements in the data flow. Here, WBS is a table with five columns, and Task Group is a set of WBS#s. More detailed definitions of the element formats and index schemes may be delayed until additional information is compiled.

PROCESS (MINI) SPECIFICATION

```
5.2 Define Schedule Process
    for each TASK in TASK GROUP
        get and list WBS# and WBS TITLE
        enter START date
        enter STOP date
        if START < STOP then print error and reenter
    end
```

DATA DICTIONARY

WBS = WBS# + Title + Start + Stop + Resources
Task Group = {WBS#}

Examples of composition techniques include the Jackson System Design and object-oriented design ("design" implying that the process steps have considerable overlap). In the Jackson System Design, the target system is represented as a discrete simulation, and the implementation is considered a set of communicating sequential processes; that is, the method allows for the modeling of the real-world environment as a computer simulation, which then is transformed into a set of sequential programs that can operate asynchronously. Conversely, object-oriented design first identifies the real-world objects that the desired system must interact with and then considers how those objects interact with each other. There are several versions of object-oriented design, but experience with its use is limited.

Programming in the Large—Design

The design process begins after there is a specification establishing what functions the software is to provide. From the discussion of analysis, we see that there is no precise division between analysis (the decision of what is to be done) and design (the determination of how to realize it). There sometimes is a contractual need to establish what the procured software is to provide, so the specification becomes part of the contract that defines the deliverable. In the essential model of the software process, however, there is continuity between analysis and design activities, and the methods often support both activities.

The basic process is one of modeling the software system and adding details until there is sufficient information to convert the design into a realization (i.e., program). Design always begins with a specification, which is a product of the analysis step. At times, the specification is a formal document establishing a set of requirements. Here, it is important to maintain traceability to ensure that all design decisions are derived from a requirement and that all requirements are satisfied in the design (i.e., there are neither extra features nor omissions). At other times (e.g., in the internal development of a product), the specification is less formal, and additional subjectivity is needed to determine that the design decisions are valid.

For any set of requirements, there are many equally correct designs. The task of the design team is to select among the alternatives those system decisions yielding a design that is, in some way, expected to be better than the others. Studies of this activity indicate that considerable domain experience is required. Also, the ability and training of the team members is some two to four times as important as any other factor in determining the cost to produce an acceptable product.

Design methods are extensions of analysis methods. For example, decomposition techniques use the DFD, and composition methods span the analysis and programming-in-the-large tasks. With decomposition techniques, "structured design" is used to model the interactions among software modules. Rules are available to guide the transition from DFDs to the "structure diagrams" depicting module control flow. As with DFDs, data dictionaries are used to describe the elements in the data flow, and the functions of the modules are detailed as "module specs" in structured English.

Other methods begin with models of the data and their temporal changes, and then derive the processes from those data structures. The Jackson Program Design, for example, models the structure of the data and then builds models of the procedures that reflect that structure. For data processing applications, there are several methods used to define the data model. One widely used method is the entity-relationship model. Here, the entities (e.g., employees, departments) and their relationships (e.g., works in) are identified and displayed graphically. Rules then can be applied to convert this conceptual model into a scheme that can be implemented with a database management system.

We have identified here many different (and often mutually incompatible) methods, but the list is incomplete. Many of those methods use some form of diagram. Most CASE tools support the DFD, structure diagram, Jackson System Design notation, and entity-relationship model. There also are proprietary tool sets that are limited to a single method. One of the benefits that any good method provides is a common approach for detailing a solution and communicating design decisions. Thus, for effective communication, an organization should rely on only a limited number of methods. The DFD and the entity-relationship model are the most broadly disseminated and, therefore, frequently will be the most practical for the communication of concepts.

Programming in the Small—Coding

Code involves the translation of a design document into an effective and correct program. In the 1970s, the concept of "structured programming" was accepted as the standard approach to produce clear and maintainable programs. The structured program relies on three basic constructs:

1. Sequence—a set of statements executed one after the other.
2. Selection—a branching point at which one of a set of alternatives is chosen as the next statement to be executed (e.g., IF and CASE statement).
3. Iteration—a looping construction causing a block of statements to be repeated (e.g., DO statement).

Every program can be written using only these three constructs. A corollary, therefore, is that the GOTO statement is unnecessary.

Structured programming introduced other concepts as well. Programs were limited to about 50 lines (one page of output). Stepwise refinement was used to guide the top-down development of a program. When a concept was encountered during programming that required expansion, it would be represented as a procedure in the user program and later refined. This method allowed the programmer to defer design activities; it also resulted in programs that were easier to read and understand. To improve comprehension, indentation and white space were used to indicate the program's structure. In time, the flow chart was replaced by the "program design language" (e.g., the minispec), which captured the program structure but omitted many program details.

Another concept introduced in the late 1970s was "information hiding," which emerged following analysis of what characteristics should bind together, what functions are retained in a module (cohesion), and how modules should interact with each other (coupling). The goal of information hiding is to yield a logical description of the function that a module is to perform and isolate the users of that module from any knowledge of how that function is implemented. Thus, the designers may alter the internal implementation of one module without affecting the rest of the program. This concept was refined and became known as the abstract data type. A data type defines what kinds of data can be associated with a variable symbol and what operators can be used with it. For example, most languages offer an integer, real, and character-string data type. The operator plus (+) has a different meaning for each data type.

With an abstract data type, the designer can specify a new data type (e.g., the matrix) and operators that are valid for that data type (e.g., multiplication, inversion, scalar multiplication). Using the terminology of the Ada[5] programming language, the abstract data type is defined in a package with two parts. The public part includes a definition of the data type and the basic rules for the operations. The private part details how the operations are to be implemented. To use the abstract data type, the programmer includes the package by name and then declares the appropriate variables to be that data type. This is an example of software "reuse." The data type operations are defined once and encapsulated for reuse throughout the software application, thereby reducing the volume of the end product and clarifying its operation.

Another technique to improve program quality is embodied in the concept of "proof of correctness," meaning that the resulting program is correct with respect to its specification. There are some experimental systems that can prove a program to be formally correct. Such systems have been used to verify key software products, such as a security kernel in an operating system. But proof of correctness usually is applied as a less formal design discipline.

"Fourth generation languages" (4GLs) represent another approach to software development. Here, special tools have been developed for a specific class of application (information processing) that facilitate the development of programs at a very high level. For example, one can produce a report simply by describing the content and format of the desired output; one does not have to describe procedurally how it should be implemented. (Thus, 4GLs generally are described as being nonprocedural or declarative.)

Validation and Verification

In the traditional descriptive flow for software development, the activity that precedes operations and maintenance is called "test." Testing is the process of detecting errors. A good test discovers a previously undetected error. Thus, testing is related to defect removal; it can begin only when some part of the product is completed and there are defects to be removed.

The validation and verification activity includes the process of testing. But it begins well before there is a product to be tested and involves more than the identification of defects. Validation comes from the Latin *validus*, meaning strength or worth. It is a process of predicting how well the software product will correspond to the needs of the environment (i.e., will it be the right system?). Verification comes from the Latin *verus,* meaning truth. It determines the correctness of a product with respect to its specification (i.e., is the system right?).

Validation is performed at two levels. During the analysis step, validation supplies the feedback to review decisions about the potential system. Recall that analysis requires domain understanding and subjective decisions. The domain knowledge is used to eliminate improper decisions and to suggest feasible alternatives. The ranking of those alternatives relies on the analysts' experience and judgment. The review of these decisions is a cognitive (rather than a logically formal) activity. There is no concept of formal correctness; in fact, the software's validity can be established only after it is in place. (Prototypes and the spiral model both are designed to deal with the analyst's inability to define a valid specification.)

The second level of validation involves decisions made within the context of the specification produced by the analysis activity. This specification describes what functions should be supported by the software product (i.e., its behavior). The specification also establishes nonfunctional requirements, such as processing time constraints and storage limitations. The product's behavior can be described formally; in fact, the program code is the most complete expression of that formal statement. Nonfunctional requirements, however, can be demonstrated only when the product is complete.

Validation and verification are independent concepts. A product may be correct with respect to the contractual specification, but it may not be perceived as a useful product. Conversely, a product may correspond to the environment's needs even though it deviates from its specified behavior. Also, validation always relies on judgment, but verification can be formalized. Finally, both validation and verification can be practiced before there is code to be tested; failure to exercise quality control early in the development process will result in the multiplication of early errors and a relatively high cost of correction per defect.

Before a formal specification exists (one that can be subjected to logical analysis), the primary method for both verification and validation is the review. In the software domain, this is sometimes called a walk-through or inspection, which frequently includes the review of both design documents and preliminary code. The review process is intended to identify errors and misunderstandings. There also are management reviews that establish decision points before continuing with the next development step. The two types of reviews have different functions, and they should not be confused or combined. Management reviews should occur after walk-throughs have been completed and technical issues resolved.

Most software tests are designed to detect errors, which sometimes can be identified by examining the program text. The tools that review the text are called "static analyzers." Some errors they can detect (such as identifying blocks of code that cannot be reached) can be recognized by compilers. Other forms of analysis rely on specialized, stand-alone software tools. "Dynamic analysis" tests, concerned with how the program operates, are divided into two categories. "White box" tests are designed to exercise the program as implemented. The assumption is that the errors are random; each path of the program, therefore, should be exercised at least once to uncover problems such as the use of the wrong variable or predicate. "Black box" tests evaluate only the function of the program, independent of its implementation.

As with equipment testing, software testing is organized into levels. Each program is debugged and tested by the individual programmer. This is called unit testing. Individual programs next are integrated and tested as larger components, which are then function tested to certify that they provide the necessary features. Finally, the full system is tested in an operational setting, and a decision is made to deploy (or use) the product. Naturally, if the software is part of an embedded system, then, at some level, the software tests are integrated with the hardware tests.

Management

We have so far emphasized the essential features of software development; that is, what makes the development process unique for this category of product. Some characteristics of the process make it difficult: the software can be very complex, which introduces the potential for many errors; the process is difficult to model in terms of physical reality; there is always a strong temptation to accommodate change by modifying the programs; and, finally, the product is always subject to change. (In fact, the lifetime cost for adaptation and enhancement of a software product usually exceeds its development cost.)

The management of a software project is similar to the management of any other technical project. Managers must identify the areas of highest risk and the strategies for reducing that risk; they must plan the sequence of project activities and recognize when deviations are imminent; they must budget time, personnel, and dollar resources and adjust these factors throughout the process; they must maintain control over the completed products; and they must establish procedures to ensure a high level of product quality.

As with any project management assignment, the manager must understand something about both the domain of application and the technology to be applied. In well-understood problem areas, this knowledge is less critical because design is reduced to the detailing of some existing design concept. But in new domains there are uncertainties, and management must be sensitive to the early resolution of high-risk problems. (This is one area in which prototyping can be most effective; another is the examination of the human interface.)

Although software engineering is a relatively new discipline, there are many tools available to help support its management. Cost-projection tools have been produced that allow a manager to build upon previous experience to estimate cost and schedule. Commercial "configuration control" systems manage the software versions and supply mechanisms to insulate programs from unauthorized or uncertified changes. Many modern program-support environments also contain tools that give management easy access to schedule and status information.

GOVERNMENT METHODS APPLIED AT APL

The approaches to software engineering taken by APL's sponsoring organizations must be considered both when an operational system is delivered and when a prototype is developed that will evolve into government specifications and procurements. For many experiments (e.g., at-sea or space-based), the whole effort is oriented toward quick deployment of existing or slightly modified sensors and support for recording, telemetry, and analysis. There are no sponsor-specified approaches, and development responsiveness often is the critical component. The software becomes a crucial integrating element. Since existing software must be modified, software engineering techniques are applied less formally.

In its work on Navy tactical systems, however, APL relies more on the use of standards. The development and acquisition of mission-critical defense-systems software is governed by DOD-STD-2167A,[6] a standard specifying a number of documents that should be generated during software development. This document-driven approach has been criticized for the lack of proper document selection and tailoring by the government and for ignoring several modern software processes or development strategies as described above. Although this standard has some drawbacks, it has provided a sound and consistent basis for software development over many years. Recently, it was modified (Revision A) to reflect better the methods available via the Ada language. This standards approach is not new; the DOD standard originated in earlier internal Navy standards.[7,8] The other main component in the Navy's approach is the use of specific programming languages. The Navy has attempted for many years to standardize languages, beginning with CS-1 in the 1960s, through the CMS-2 and the introduction of Ada.

The Navy also has standardized computer hardware, in particular, the AN/UYK-20, and more recently the AN/UYK-44 and AN/AYK-14, which are the 16-bit common instruction set standards. The AN/UYK-7 is upward-compatible with the newer 32-bit AN/UYK-43 standard computer. For the Navy to support those computers with the new Ada language, the Ada Language System/Navy project is tasked to develop production-quality Ada translators (first released in June 1988) for the AN/UYK-44, AN/AYK-14, and AN/UYK-43. Considerable emphasis has been placed on the support of fast interrupts, a requirement of many Navy embedded systems. Within a common command language framework, there are tools for version and database maintenance, software-problem-report tracking, documentation updating, and report generation.

The Ada language has matured, and the translators are rapidly becoming viable production tools. Some recent benchmark programs executing on the MC 68020 processor indicate that the Ada code is almost as fast as code generated by popular C-language translators. Ada is now a workable language for military use, and the benefits of the object-oriented approach can be obtained from specification through operational code and, most important, during evolution and maintenance. APL has explored Ada applications for several tactical systems. Whenever a mature Ada translator and target run-time system have been available, the projects have been successful. One example is the redesign and reimplementation of a simplified version of the Aegis Command and Decision System.[9] This project, while successful, also identified several problems with the 1985 Ada technology. Many of those problems have disappeared with more mature Ada products. In follow-on projects, the reuse of both internally developed and externally available Ada modules has been successful. Also, Ada module integration has been faster, and fewer errors were detected than with other languages.

Most software development at APL involves the support of experimental or "one of a kind" systems. Although these systems frequently require the proper use of software engineering techniques, they typically are not controlled by the sponsor as is a mission-critical Navy system (i.e., DOD-STD-2167A). This offers an opportunity to try new techniques. For example, in some satellite altimeter work for NASA, the DeMarco and Yourdon methodology, with Ward/Mellor real-time extensions,[10] is being successfully applied using new CASE tools. The recent APL projects requiring Ada have used the Bahr object-oriented design approach,[11] which (with some modification) has been particularly useful when the target language is known to be Ada. Some projects derived their approach from a combination and tailoring of DOD-STD-2167A and emerging IEEE software standards. This approach provides guidelines and templates, and helps to alleviate problems arising from staff reassignment and short corporate memory.

ROADS TO IMPROVEMENT

There are many possible roads to improvement for the software engineering process, with numerous feeders and intersections, and some dead ends. Three promising routes are standards and guidelines, technology centers, and research. We focus below on the first two. (Research is covered in the next section.)

Standards and Guidelines

The primary roles of international standards and guidelines in the software arena are to provide application portability by defining interfaces and to define the state of practice. The international standards process is well established. Standards usually deal with what should be done, not how. This is sometimes expressed as the public view (not the private view). When standards deal with interfaces, the "how" often is stated, simply because it is a necessary part of the public view.

Establishing international standards is a time-consuming process that depends on the interplay among industry, government, and professional societies. In the national and international forum, it is a consensus-building process. A typical IEEE standard takes about five years to progress through its various stages. Since most standards are based on experience, the impression frequently is created that standards institutionalize past practices and stifle new technology. Exclusive of military environments, the major players in international software standards development are the IEEE, the International Standardization Organization, and the American National Standards Institute. It is not possible in this article to describe all the international software standards efforts and their interactions. A summary of 11 IEEE standards and 13 standards projects is available, however.[12] Significantly, software standards play an ever-increasing role in establishing the state of software engineering practice and in interfacing various software processes.

The primary Defense Department standard for software development is DOD-STD-2167A. The IEEE and other military standards cover similar subjects (e.g., documentation, quality assurance, audits). One major distinction exists. The military standard deals mainly with contract deliverables; thus, document content is emphasized. The IEEE standards capture more of the essence of the software development process. Additional Navy standards traditionally have been stated in different terms—the use of standard, government-furnished equipment and software.

Technology Centers

Technology centers such as the Software Engineering Institute (SEI) and the Software Productivity Consortium (SPC) have been established by the government and industry to improve software engineering technology transition. A third center, the Microelectronics and Computer Technology Corporation (MCC), has a broader mission that also includes software engineering.

The SEI is a federally funded research and development center operated by Carnegie–Mellon University under contract with the U.S. Department of Defense. Its mission is to (1) bring the ablest professional minds and the most effective technology to bear on rapidly improving the quality of operational software in mission-critical computer systems; (2) bring modern software engineering techniques and methods into practice as quickly as possible; (3) promulgate the use of modern techniques and methods throughout the mission-critical systems community; and (4) establish standards of excellence for software engineering practice.

Software technology transition is SEI's major focus. Unlike industrial consortiums, software engineering research is not a significant part of its mission. The institute has convened several workshops over the last few years, and many organizations (including APL) in industry, academia, and government have affiliated with it. As a result, SEI has projects in several areas, including technology surveys, course and textbook development, software reliability techniques, uniform communication medium for Ada programs, documentation and reporting conversion strategy, software process modeling, software maintenance, human interface technology, legal issues, environments, and pilot projects.

Several course modules for undergraduate and graduate software engineering courses have evolved from the institute's software engineering education program. The recently introduced master's degree course entitled "Projects in Software Engineering" at the APL Center of The Johns Hopkins University Continuing Professional Programs is based on SEI material. A full description of the university's professional computer science curriculum was presented at a recent SEI education workshop.[13] Besides interesting technical and education material, the institute has produced guidelines for program managers on the adoption of Ada[14] and a method for evaluating an organization's software engineering capability.

The SPC was formed in 1984 by 14 companies to close the mission-critical software gap between defense and aerospace system hardware and the availability of software to drive those systems. Its goal is to build the software tools and techniques needed to accelerate the software engineering cycle, improve the quality and functionality of the final software product, and make major systems software easier and less expensive to maintain.

The objective of SPC's technical program is to create a dramatic increase in the software productivity of member companies. Over the next five years, the consortium will develop and provide members with a range of software engineering products and technologies for an integrated development environment. The 14 partners receive exclusive rights to products, technologies, research, and support services developed by SPC.

The products will exploit three key concepts: symbolic representation, prototyping, and reusable components. Symbolic representation makes the process of developing software a "tangible" activity by representing software life-cycle objects in more "natural" ways to the working engineer. (The concepts of prototyping and reusable components are discussed elsewhere in this article.) Within the context of an integrated development environment and project libraries, SPC will develop tools for building requirement and design specifications and related code, synthesizing prototypes, performing dynamic assessments, and managing software development projects. At a recent meeting of companies developing and marketing CASE tools, SPC launched an initiative to establish an industrywide consensus on effective tool-to-tool interface standards. Those standards will represent the first steps in building an integrated environment.

MCC was established by 21 shareholder companies in 1983. The consortium has several research programs, ranging from semiconductor packaging to software technology. Each program is sponsored by a subset of participating shareholder companies.

The software technology program focuses on the front end, upstream in the software cycle, where little research has been performed. This program has created a computer-aided software design environment call Leonardo. The environment is to concentrate on requirements capture, exploration, and early design. Academic research has focused on downstream activities, where formalism and automation are more obvious. MCC's research emphasis is on defining and decomposing a large problem into smaller problems and on selecting algorithms, and is geared to teams of professional software engineers working on large, complex systems. Researchers are working on Leonardo architecture, and three components: complex design processes, a design information base, and design visualization.

The corporation does not consider its research complete until it is put to use by the sponsoring companies. Also, MCC believes it is easier to transfer and establish tools and make them consistent than to transfer methodologies and methods.

A VIEW TO THE FUTURE

We began this article with a review of how software differed from hardware and noted that—once the technical manager understands the software process—the management of software is much like that of hardware. We then described the software process, characterized more by variety than by clarity and consistency. Despite our significant accomplishments with software, there remain conflicting methods, limited formal models, and many unsubstantiated biases. But we present below some significant trends in the field.

Formalization

Some new paradigms extend the formalism of the programming language into an executable specification. A specification defines the behavior for all implementations. An executable specification does not exhibit the intended behavior efficiently, but a program is an implementation of that behavior, optimized for a specific computer. We see this because there are systems that we know how to specify exactly, but we do not know how to implement them efficiently. For example, one can specify what a chess-playing program should do without being able to describe an efficient implementation. The hope is that the executable specification will supply a prototype for experimentation that ultimately can be transformed into an efficient program. But the concept has not been demonstrated outside the laboratory, and it is not clear how this process model can be managed.

Automatic Verification

In the discussion of validation and verification, we noted that proofs (verification) could be objective only when the parent specification was clear (formal). There is, therefore, considerable interest in working with mathematically formal specifications at an early stage in the design process, since it then would be possible to prove that each detailing step was correct with respect to this high-level source. A theorem prover could be used to automate the process. Testing then would not be necessary, because no errors would exist. Naturally, validation still would be required.

Automated Tools and Environments

There is considerable interest in the use of CASE tools and in program support environments. Unlike the formalisms addressed above, most tools and environments are commercial products that implement techniques developed in the mid-1970s. Thus, these tools and environments respond to a marketplace demand and provide a means for making the application of current practice more efficient. Their primary benefit is one of reducing the manual effort and thereby the number of errors introduced. As new paradigms are introduced, this focus may limit their use or force changes in the approaches taken.

Artificial Intelligence and Knowledge Representation

Although there are many definitions of artificial intelligence and debates about what it accomplishes, it has had an impact on our perceptions of what computers can do and how to approach problems. The software process is one of representing knowledge about a problem in a way that facilitates its transformation (detailing) into an implementable solution. Thus, there are many software engineering methods and tools that owe their origins to artificial intelligence. Some projects, such as the development of object-oriented programming, have been successful and are available to developers; many others still are in the research stage. One can expect that a knowledge orientation to the problem of software design will have considerable impact.

New High-Order Languages

The major advances of the 1960s can be attributed to the use of high-order languages, but it is doubtful that current language improvements will have much impact on productivity. Many proven modern programming concepts have been incorporated into Ada, and the commitment to this language clearly will familiarize developers with those concepts and thereby improve both product quality and productivity. At another extreme, 4GLs offer tools to end users and designers that, for a narrow application domain, yield a tenfold improvement in productivity at a price in performance. Still, neither high-order languages nor 4GLs can match the improvements we are witnessing in hardware cost performance.

Software Reuse

The concept of software reuse was first perceived in the context of a program library. As new tools have been developed, the goal of reusable components has expanded. For example, Ada packages that encapsulate abstracted code fragments can be shared and reused. The artificial-intelligence-based knowledge perspective also suggests ways to reuse conceptual units having a granularity finer than code fragments and program libraries. Finally, the extension of 4GL techniques provides a mechanism for reusing application-class conventions with a natural human interface.

Training and Domain Specialization

All software development requires some domain knowledge. In the early days of computing, the programmer's knowledge of the new technology was the key, and the domain specialist explained what was needed. Today, almost every recent college graduate knows more about computer science than did those early programmers. Thus, there is an emphasis on building applications. As more tools become available, one can expect software developers to divide into two classes. The software engineer will, as the name implies, practice engineering discipline in the development of complex software products, such as embedded applications and computer tools for end users. The domain specialists will use those tools together with the domain knowledge to build applications that solve problems in their special environment. We can see this trend in the difference between Ada and the 4GLs. Ada incorporates powerful features that are not intuitively obvious; the features are built on a knowledge of computer science and must be learned. The 4GLs, however, offer an implementation perspective that is conceptually close to the end user's view. The software engineer builds the language; the domain specialist uses it.

WHAT OTHERS SAY

What do the experts in software engineering say about the future of this discipline and the hope for significant improvements in productivity? In explaining why the Strategic Defense Initiative is beyond the ability of current (and near-term) software practice, Parnas[15] offered a negative critique of most research paths. He said that the problem involves complex real-time communication demands, adding that there is limited experience in designing programs of this architecture and magnitude and that there is no way to test the system thoroughly. No ongoing approach, he concluded, could overcome these difficulties.

Boehm,[1] in an article on improving productivity, was more positive. Speaking of state-of-the-art software applications, he offered this advice: write less code, reduce rework, and reuse software—especially commercially available products, where possible.

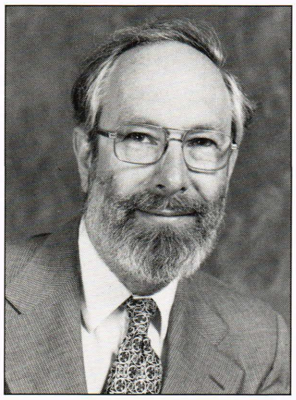
Brooks[16] discusses the possibility of improving software productivity. He has catalogued research directions in some detail and concluded that the biggest payoff would come from buying rather than building, learning by prototyping, building systems incrementally, and—most important to him—training and rewarding great designers. Of those four recommendations, the first reflects the spinning off of tools that can be used by domain specialists, and the next two relate to the need to build up knowledge about an application before it can be implemented. The last of Brooks's positive approaches recognizes that software design (like every other creative activity) depends on, and is limited by, the individual's ability, experience, understanding, and discipline.

REFERENCES and NOTES

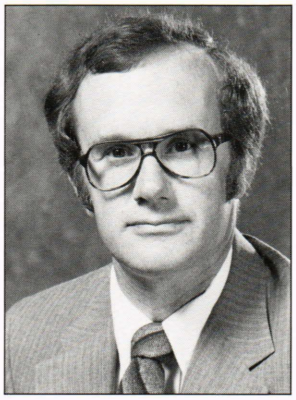
1. B. W. Boehm, "Improving Software Productivity," *IEEE Computer* **20**, 43–57 (1987).
2. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* **21**, 61–72 (1988).
3. B. W. Boehm, "Industrial Software Metrics Top 10 List," *IEEE Software*, 84–54 (Sep 1987).
4. Two books that are highly recommended are R. Fairley, *Software Engineering Concepts,* McGraw-Hill, New York (1985) and R. Pressman, *Software Engineering: A Practitioner's Approach,* 2nd ed., McGraw-Hill, New York (1987).
5. Ada is a registered trademark of the U.S. Government, Ada Joint Project Office.
6. DOD-STD-2167A, "Military Standard Defense System Software Development," (29 Feb 1988).
7. MIL-STD-1679 (Navy), "Military Standard Weapon Software Development," (1 Dec 1978).
8. SECNAVINST 3560.1, "Tactical Digital Systems Documentation Standards," (8 Aug 1974).
9. D. F. Sterne, M. E. Schmid, M. J. Gralia, T. A. Grobicki, and R. A. R. Pearce, "Use of Ada for Shipboard Embedded Applications," Annual Washington Ada Symp., Washington, D.C. (24–26 Mar 1985).
10. S. J. Mellor and P. T. Ward, *Structured Development for Real-Time Systems,* Prentice-Hall, Englewood Cliffs, N.J. (1986).
11. R. J. A. Bahr, *System Design With Ada,* Prentice-Hall, Englewood Cliffs, N.J. (1984).
12. G. Tice, "Looking at Standards from the World View," *IEEE Software* **5**, 82 (1988).
13. V. G. Sigillito, B. I. Blum, and P. H. Loy, "Software Engineering in The Johns Hopkins University Continuing Professional Programs," 2nd SEI Conf. on Software Engineering Education, Fairfax, Va. (28–29 Apr 1988).
14. J. Foreman and J. Goodenough, *Ada Adoption Handbook: A Program Manager's Guide,* CMO/SEI-87-TR-9, Software Engineering Institute (May 1987).
15. D. L. Parnas, "Aspects of Strategic Defense Systems," *Commun. ACM* **12**, 1326–1335 (1985).
16. F. P. Brooks, "No Silver Bullet," *IEEE Computer* **20**, 10–19 (1987).

ACKNOWLEDGMENTS—The authors gratefully acknowledge the very helpful suggestions of J. E. Coolahan, M. J. Gralia, R. S. Grossman, and J. G. Palmer.

THE AUTHORS



BRUCE I. BLUM was born in New York City. He holds M.A. degrees in history (Columbia University, 1955) and mathematics (University of Maryland, 1964). In 1962, he joined APL, where he worked as a programmer in the Computer Center. During 1967-74, he worked in private industry, returning to APL in 1974. His special interests include information systems, applications of computers to patient care, and software engineering. From 1975-83, he served as director of the Clinical Information Systems Division, Department of Biomedical Engineering, The Johns Hopkins University.



THOMAS P. SLEIGHT received his Ph.D. from the State University of New York at Buffalo in 1969. Before joining APL, he spent a year as a postdoctoral fellow at Leicester University, England. At APL, Dr. Sleight has applied computers to scientific defense problems. He has served as computer systems technical advisor to the Assistant Secretary of the Navy (R&D) and on the Ballistic Missile Defense Advanced Technology Center's Specification Evaluation Techniques Panel. He has participated in the *DoD Weapons Systems Software Management Study*, which led to the DoD directive on embedded computer software management. Dr. Sleight served as supervisor of the Advanced Systems Design Group from 1977-82 in support of the Aegis Program and the AN/UYK-43 Navy shipboard mainframe computer development and test program. Since 1982, he has served in the Director's Office, where he is responsible for computing and information systems.