Appendixes

`

Appendix A

Technology for Producing Dependable Software

Introduction

Chapter 9 of this report often refers to the technology of specific phases in the software life cycle. The application of such technology is known as *software engineering. This* appendix describes the state of the art in software engineering and prospects for improvements in the state of that art. It serves as a tutorial for those unfamiliar with Department of Defense (DoD) software development practices. It provides supporting detail for the discussions in chapter 9 of the technology available for producing dependable systems.

Origins of Software Engineering

The term "software engineering" originated in 1968. Around that time, computer scientists began to focus on the difficulties they encountered in developing complicated software systems. A recent definition of software engineering is:

> . . . the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation.[1]

Another recent definition adds requirements for precise management and adherence to schedule and cost:

> Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.[2]

The Institute of Electrical and Electronic Engineers (IEEE) defines software engineering as:

> The systematic approach to the development, operation, maintenance, and retirement of software.[3]

Before the late 1960s, managers paid little attention to the systematizing of software construction building. Most software systems were not complicated enough to occupy large numbers of people for long periods of time. Existing computers were not big or fast enough to solve very complicated problems:

> As long as there were no machines, Programming was no problem at all; when we had a few weak computers, Programming became a mild problem, and now that we have gigantic computers, Programming has become an equally gigantic problem.[4]

Software engineering technology has improved since 1972, but not as quickly as the capabilities of computers. Studies in software engineering technology transfer show that ideas typically take 18 years to move from research environments, such as universities and laboratories, to common uses During this time, considerable experimentation and repackaging occur.

Advances in software engineering technology often take the form of better techniques for program design and implementation. Some techniques demand no more than a pencil and paper and an understanding of their concepts. Most, however, become partially or fully automated. The form of automation is generally a computer program, known as a "software engineering tool." One example is the compiler-which helps to debug other programs; another is a program that checks the consistency of software specifications. Because software engineering tools themselves take the form of complex computer programs, they are subject to the typical problems involved in producing complex, trustworthy software. This fact helps explain why software engineering technology lags hardware engineering technology.

The trend toward use of software engineering tools seems to be growing, as evidenced by such projects as:

- DoD's Software Technology for Adaptable, Reliable Systems (STARS), whose purpose has been to produce an integrated set of tools for DoD software engineers;

---

[1] Barry W. Boehm, *Software Engineering Economics* (Englewood Cliffs, NJ: Prentice-Hall, 1981), p. 16.

[2] Richard Fairley, *Software Engineering Concepts, (New* York: McGraw-Hill, 1985), p. 2.

[3] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 729-1983.

[4] Edsger W. Dijkstra, "The Humble Programmer," *Communications of the ACM*, 15(10):859-866, *1972.*

[5] William E. Riddle, "The Magic Number 18 Plus or Minus Three: A Study of Software Technology Maturation," ***ACM Software Engineering Notes,*** vol. 9, No. 2, 1984, pp 21-37.

The figure of 18 years for technology transfer is consistent with other engineering fields during periods of technological innovation, as analyzed in Gerhard O. Mensch, ***Stalemate in Technology*** (Cambridge, MA: Ballinger, 1979).

- the Software Productivity Consortium, formed to produce software engineering tools for its client members, including many of the nation's largest Aerospace companies; and
- the Microcomputer and Electronics Consortium, one of whose purposes is to produce better software engineering technology for its client members.

State of the Art in Software Engineering

This section discusses the current state of the art in software engineering technology. It considers the application of that technology to systematic approaches to software development. Finally, it reviews recent proposals for improvements in software engineering to aid in the development of ballistic missile defense (BMD) software.

The Software Development Cycle

The process of developing and maintaining software for military use is described in DoD military standards documents as the "software life cycle." The description here is simplified for the purposes of this report. The activities described are common to nearly all DoD projects, though they vary in the amount of attention paid to each, the products produced by each, and the number and kind of subactivities in each.

Furthermore, the initial set of activities described encompasses only development, up to the point of acceptance of the system by DoD. Our description lumps activities following development into the category of "maintenance," which is discussed in a separate section. Finally, the activities described here generally conform to the model of development set forth in the DoD Standard (No. 2167) for software development. Commercial development and advanced laboratory work may follow considerably different procedures (although all tend to produce documentation similr in intent to that described in the following sections).

Feasibility Analysis

The first phase of development is an analysis of the DoD's operational needs for the proposed system. This phase may start with a series of studies of the feasibility of meeting those needs. The reported results of the feasibility studies are often based on computer simulations of the situations that the system would have to handle. The environment in which the system would operate may be characterized in terms of quantifiable parameters. This process is analogous to telephone company analyses of the traffic load–the number of calls per hour expected at different times of the day, week, and year-to be placed on a new switching center. The feasibility analysis may be performed on a contract basis by systems analysts who are not software engineers. In the case of the Strategic Defense Initiative (SDI), systems analysts who are familiar with BMD have done much of the feasibility analysis as part of the competitive system architecture contracts (see ch. 3).

Feasibility analysis sometimes includes constructing a software prototype designed to investigate a few specific issues, for example, what the mode of interaction between human and computer should be or which tracking algorithms would work best under different circumstances. Most system functions are not implemented in prototypes. To save time, the development of prototype software does not follow the standard cycle. Software in the prototype is not usually suitable for reuse in the actual system. The prototype is usually discarded at the end of the software development cycle.

The end-product of the feasibility analysis is a document that describes what functions the new system needs to perform and how it will work. This report is sometimes called a "concept of operations" or an "operational requirements" document. The operational requirements document will form the basis for a request for proposals to potential development contractors. Once the contract has been awarded, the document will underlie the statement of requirements to be met by the system—that is, the description of what the contractor must build. The (SDI) battle management software development project is currently in the feasibility analysis stage.

Software Requirements Analysis and Specification

Software engineers enter the software development process at the software requirements analysis and specification phase. From their interpretation of the operational requirements document, they write a requirements or performance "specification" for the software (including how it is to interface with other systems). Upon government approval, this specification document supplies the contractual criteria of acceptability of the software to the government. The specification differs from the operational requirements as follows:

- It is more detailed than the operational requirements statement. Where the latter may

vaguely describe the functions to be performed, such as "simultaneously track 10,000 missiles," the requirements specification will describe a sequence of subfunctions needed to implement the operational requirement. The description of each subfunction will include a description of the input supplied to the subfunction, the output produced by it, and a brief explanation of the algorithm by which the inputs are transformed into outputs.

For example, a subfunction of tracking may be to update the track for a particular missile. The inputs of that subfunction are the current file of tracks and a new track report. Its output is an updated track file. The algorithm might consist of the following steps:

—retrieve the existing track for the missile from the track database;
—update the position, velocity, acceleration, and time of last report components of the missile track; and
—store the updated track entry back into the database.

- It describes the interfaces between the new system and all the other systems with which it must interact. Sometimes these interfaces are described in a separate document.
- It describes the interfaces to the hardware devices that the software must control, such as weapons, or from which the software must obtain information, such as navigation devices like inertial guidance units.

When the requirements specification is complete, the government holds a "requirements review" at which it decides what, if any, changes must be made. The specification becomes a contractually binding document and is passed on to the software designers. The review is the last time at which requirements errors can be corrected cheaply: few assumptions have yet been made about the way the requirements will be implemented as computer programs. Procedures known as "configuration control" are established. These ensure that the specification is not arbitrarily changed. The specification becomes the "baseline requirements specification." All further changes go through a formal approval cycle, requiring the concurrence of a committee known as a "configuration control board."

Design

The purpose of the design phase is to produce a "program design specification" of how a computer program can be written to satisfy the requirements specification. The design specification usually describes the division of the software into components. Each component may be subdivided again, with the subdivision process eventually ending in subcomponents that can be implemented as individual subprograms or collections of data. The components resulting from the first subdivision, sometimes called "configuration items," are used to track the status of the system throughout its lifetime.

The organization that emerges from the design process is known as the structure of the software, and the criteria used are called "structuring criteria." The relationship among components depends on the criteria used in the subdivision process. For example, if the criterion for subdivision is function, at the first subdivision a component called "tracking" might be formed. At the second level one might find tracking subdivided into functions such as "obtain object track" and "update object track." Such a subdivision is called a "functional decomposition."

A very different criterion is type of change. At the first level one might then see components such as all decisions that will change if the hardware changes and all decisions that will change if software requirements change. At the second level one might find hardware decisions subdivided into decisions about sensor hardware, decisions about weapons hardware, and decisions about computer hardware. Such an organization is called an "information hiding decomposition."

The structuring criteria are key to understanding the design and the trade-offs it embodies.[6] Those who use functional decomposition argue that it results in software that performs more efficiently. Those who use information hiding argue that it results in software that is easier to maintain because it is easier to demonstrate correct and easier to understand. Other criteria optimize for other factors, for example, fault-tolerance, security, and ease of use. As might be expected, there is no single criterion that simultaneously optimizes all design goals.

An important purpose of the design specification is to describe the "interfaces" among components. The interfaces consist of the data to be passed from one component to another, the se-

[6] Good designers find it useful to structure the design in several different ways to permit study of different trade-offs. For example, an information-hiding decomposition is useful in optimizing for changes later in the life-cycle of the software. A functional decomposition helps ensure that all functions are performed and makes it easy to analyze the efficiency of software. An important problem for the designer is to be able to represent the different design structures so that they are consistent with each other.

quences of events to be used in coordinating the actions of the components, and the conditions under which the components interact. Once the interfaces are established, individual design teams may design each component. Since each component may interact with several others, agreement on the interfaces is crucial for effective cooperation among the design teams.

Such agreement is equally important for those who must later implement the design as a computer program. A mistaken assumption about an interface will result in an error in the software and a failure in the operation of the software. A change in an interface requires agreement among all those working on the interfacing components, and may result in redoing weeks or months of work. Getting the interfaces right is generally agreed to be the most difficult part of developing complex software.

During the design process several reviews of the design are held. The purpose of the reviews is to ensure that the design is feasible and correctly implements the requirements. Early reviews on will be a "preliminary design review." When the designers feel that the design specification is sufficiently complete to be turned into computer programs, they hold a final, "critical design review. " Errors found at this point can still be corrected relatively cheaply. Once they become embedded in programs they are very much more difficult to find and correct. Each design review results in changes to the design specification. Once the changes are completed, the design specification becomes the basis for producing computer programs. It is then placed under configuration control, much as the approved requirements specification is.

Code

The process of translating the program design specification into computer programs is known as "coding." By DoD policy, command and control systems, weapons, and other software development projects must use a DoD standard programming language.[7] Individual programmers work from the design specification to implement the components as computer programs. The more completely and precisely the components and their interfaces are defined, the less communication is required among the programmers. They can then work independently, in parallel. An incomplete or ambiguous specification requires the programmers to make design decisions, often with incomplete and unrecorded communication among each other.

Programming is writing instructions for a computer to perform a function described in the design specification. The instructions are packaged together as a subprogram or a set of subprograms that cooperate to perform the function. Before it can be executed, a subprogram must be translated by a compiler from the programming language into machine language. Part of the programming job is to devise and perform tests on each subprogram to show that it works properly. A programmer usually goes through several cycles of writing, testing, and revising a subprogram before he is ready to declare it finished. When a programmer is satisfied that his subprograms perform correctly, he submits it to a test group.

Test

A separate group has the sole responsibility to devise, perform, and report on the results of tests. With no knowledge of the design, this group devises tests based on the requirements specification. It sends components that fail tests back to their developers with descriptions of failures and no attempts to diagnose the reasons for failures.

Test performance is the primary basis for confidence (or no confidence) that a system behaves as it is supposed to. A variety of techniques tests trustworthiness, fault-tolerance, correctness, security, and safety. It is during testing that components of the system first operate together and as a whole. The following sections describe the steps in the integration and testing process. At each step, each of the reliability aspects maybe tested.

The test process resembles a reversal of the design process. Subprograms are first tested individually, then combined into components for integration tests. Components are integrated again and tested as larger components, the process continuing until all components have been combined into a complete system. To a large extent, integration testing may be thought of as testing the interfaces between components. It is in integration testing that mistaken assumptions about how other programs behave first manifest themselves as failures.

Early tests in the process often include supplying erroneous input data to components or placing them under atypical operating conditions. Such conditions might include heavy computational loads or undesired events that, while abnormal, might occur. Such "stress" tests are designed to find out how fault-tolerant the system will be.

---

[7]Current DoD policy mandates the use of Ada as a standard programming language, unless the developer of a system obtains a waiver. Prior to the advent of Ada, each service had its own standard programming language.

As integration progresses, the total number of possible states of the formed component is the product of the number of states of its constituents. Combining component A with N states and component B with M states results in component C with M times N states. If R tests were performed on A, and T tests on B, combining each test of A with each test of B would require performing R times T tests on C. For large systems, it is impractical to perform the number of tests needed: at each integration stage the number of tests performed relative to the number of possible states becomes quite small. The tests performed on the entire integrated system include only a few typical expected scenarios.

The integration procedure described above is called "bottom-up," since it starts on the lowest component level and proceeds upward. A second integration technique that is becoming more common starts with top-level components. It attempts to test interfaces as early as possible. This technique requires the writing of dummy subcomponents that simulate only some of the actions of the actual future subcomponents, but that use the same interfaces. The dummies, called "stubs," are gradually replaced with the actual subcomponents as system "top-down" integration proceeds.

When the developer deems the system ready for delivery, a contractually-specified formal test procedure is performed to ensure that the system is acceptable to the government. This "acceptance test" consists of running several scenarios, and it may test endurance and handling of stress. Acceptance tests are performed under government observation under conditions as closely approximating real use as possible. If included, an endurance test consists of continuous simulated use of the system for a minimum of 24 hours. Endurance tests are important for systems that are expected to operate continuously once placed in service. Once a system has passed its acceptance tests, it is delivered to the government and enters the remaining phase of its life cycle, known as maintenance.

Despite elaborate test procedures, all complex software systems contain errors when delivered. As previously noted, software tests cannot be exhaustive and cannot be relied upon to find all errors. As an example, during the operational evaluation of the AEGIS system on the U.S.S. *Ticonderoga,* **20** target missiles were fired while the ship was at sea under simulated battle conditions. Some of the target missiles were fired simultaneously into the area scanned by the AEGIS combat system; thus, the 20 targets constituted fewer than 20 scenario tests. The tests revealed several software errors, costing approximately $450,000 to fix.[8]

Since errors do remain in software after acceptance testing, the correction of errors continues as a major activity after a system has been delivered and put into use.

Maintenance

Unlike hardware, software contains no physical components and does not wear out as a result of continuing use. Maintenance is really a misnomer when applied to software:

> In the hardware world, maintenance means the prevention and detection of component failure caused by aging and/or physical abuse. Since programs do not age or wear out, maintenance in the software world is often a euphemism for continued test and debug, and modification to meet changing requirements.[9]

Errors emerging during system use must be corrected, and the system must be retested to ensure that the corrections work properly and that no new errors have been introduced. Correcting an error may entail reanalyzing some requirements and doing some redesign; it almost certainly demands rewriting some code. Accordingly, all of the development activities also occur during maintenance.

Even when requirements appear to be complete and consistent, as users gain experience with a system they may change their minds about the performance they desire from it. The automobile is a case in point. Drivers' behavior and expectations about the performance of their vehicles changed as new possibilities for travel emerged and as new technology became available. Behavior also changed as economic and political situations changed. For example, wartime conditions affected the cost and availability of cars and auto parts, and oil production decisions affected the price of fuel. Similarly, as writers have switched from typewriters to word processors, both their writing habits and the features they expect in a word processor have changed.

Maintenance costs are now becoming the major component of software life-cycle costs. Some data show that by 1978, 48.8 percent of data processing costs were spent on maintenance activities.[10]

---

[8] Discussion of the results of the Ticonderoga operational evaluation testsmaybe found in U.C. Congress, Senate Committee on Armed Servicesthe record of the hearings before the Committee on Armed Services, United States Senate, 98th Congress, second session on S.2414 part 8, Sea Power and Force Projection, *Mar.* 14, 18, 29, *Apr. 5, 11*, May 1, 1984.

[9] D. David Weiss, U.S. Naval Research Laboratory, "The MUDD Report: A Case Study of Navy Software Development Practices," NRL Report 7909, May 1975.

[10] Boehm, *Software Engineering Economics*, op. cit., footnote 1, figure 3-2, p. 18.

New technology, new strategies, and new computational algorithms would cause envisioned BMD systems to evolve over many years, Their long-lifetime, complexity, and evolutionary nature will magnify the general trend towards relatively larger software maintenance costs.

Interaction Among the Phases

In the preceding sections the different phases of the software life cycle are described as if they occur in a strict sequence. In fact, there is considerable feedback among the phases. Changes in requirements, design, and code occur continually. Large systems may be subdivided into subsystems, for each of which there is a separate requirements specification and a separate development cycle. These separate developments may proceed in parallel or sequentially.

The first planned delivery in a sequential development is called the initial operating capability (IOC). Sequencing the delivery of different versions of the system over time permits faster delivery of some capabilities, but it introduces additional problems into all of the development phases. The system must be designed so that added capabilities do not require large changes to existing design or code. In particular, interfaces must be designed to take into account potential future changes in capabilities. This is another example of a problem in large, complex systems that does not occur in small systems, where the entire system is delivered at once.

The requirements analysis and design phases usually consume about 40 percent of the development effort, the coding phase about 20 percent, and the testing phase about 40 percent. For long-lived systems, the maintenance phase consumes 60-80 percent of the total lifecycle cost. For this reason, the trend in large, long-lived systems is to try to develop the software so as to make the maintenance job easier. Since the principal activity in maintenance is change, an important development consideration is how to make change easier.

Software Engineering Technology

Software engineering technology research and development tend to focus on particular phases of the software life cycle. For example, work on improving design techniques is often independent of work on improving techniques for translating high-level languages to machine code. One reason is that the different phases present very different problems for the software engineer.

The next few sections briefly describe the state of the art, the state of the practice, and the direction in which software technology is currently moving, particularly within DoD. Much of the work in the last few years has concentrated on creating automated support for software development techniques. Such support usually consists of one or more programs, called tools.[11]

Constructing Prototypes

During the feasibility analysis and requirements specification stages, software engineers sometimes quickly produce prototypes to help feasibility analysis and to explore different ways in which users might interact with the completed system. Such *rapid prototype are* intended to allow exploration of only a few issues: they are not intended to be models of the final software. They are often executed in different languages, on different computers, and using a different development process than the final software. Usually less than 10 percent of development effort is spent on such prototypes.

Sometimes software engineers do full prototyp*ing.* Full prototyping means building a complete prototype system and then discarding it. This approach has been advocated by Brooks:

> Inmost projects, the first system built is barely usable. It may be too slow, too big, too awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. . . . all large-system experience shows that it will be done. Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time....
>
> Hence plan to throw one away: you will, anyhow."[12]

Specifying Requirements

Software requirements specifications constitute an agreement between the customer and the software developers. In stating what the software must do, the specifications must be unambiguous, precise, internally consistent, complete, and cor-

[11] Rather than trying to describe even the few most notable examples, this appendix just indicates general trends. For a more detailed survey of software technology that maybe applicable to SDI, the reader should see the following Institute for Defense Analysis report on the subject prepared for the SDIO: Samuel T. RedWine, Jr., Sarah H. Nash, et al., *SDI Preliminary Software Technology Integration Plan,* IDA paper P-1926, July 1986.

[12] Frederick P. Brooks, Jr., The *Mythical Man-Month: Essays on Software Engineering* (New York, NY: Addison-Wesley, 1975).

rectly representive of the customer's desires and developers' intent. By systematizing the process, methodologies for analyzing and expressing requirements are intended to help the analyst be complete, consistent, and clear. Some tools partly automate the process—sometimes by providing mechanisms to support a particular methodology, sometimes just by providing storage and retrieval of documentation.

Before 1975, nearly all requirements were manually produced. Although they might contain mathematical equations, they used no formalisms or notations tailored to the job of writing software specifications. The underlying methodology focused on describing functions the software had to perform and specified the input to and the output from each function. By about 1980, two or three new methodologies had appeared, incorporating novel methods of decomposition and corresponding notations and formalisms.

Also appearing were several tools representing somewhat clumsy attempts to automate the processes of storing and retrieving requirements specifications and of performing internal completeness and consistency checks. More advanced tools have added some simulation capability, enabling the requirements analyst to run a simulation of his system based on the description stored by the tool. The early tools enjoyed a brief popularity that has not been sustained.

Recent technology includes attempts to combine automated support for methodologies with microcomputer systems, resulting in so-called "software-engineering workstations." Simpler workstations may use word processors to automate the text maintenance and production process. More complex workstations use document control systems on minicomputers to manage the entry, maintenance, and production of requirements documentation; such systems may feature version control and graphics support. The more tedious jobs of producing and maintainingg text have been automated, but the more difficult jobs of assuring that requirements are complete, consistent, and feasible to implement have not yet been much affected.

Design

Design technology is in a similar state to that of requirements specification. There is still little agreement on the appropriate techniques for representing and specifying designs. A few design methodologies have become popular in the last 10 years, and there are a few supporting tools that help the designer. There are a few serious attempts to integrate requirements specification and design support technology, but they have not been very successful. DoD has concentrated on finding design techniques that are compatible with Ada (the recently adopted standard DoD programming language), then developing tools that support those techniques.

Recent software design technology is on a par with requirements specification technology: the development of workstations and personal computer tools aimed at supporting the designer's job has followed the development of similar tools on larger computers.

Validating and Verifying Requirements and Design

According to one estimate, errors in large projects are 100 times more expensive to correct in the maintenance phase than in the requirements phase.[19] Supporting data suggest that the relationship between the relative cost to fix an error to the phase in which the error is detected and corrected is exponential. Accordingly, products of the software development phases undergo some kind of validation and verification several times during each phase and at the end of each phase.

Although simulation is used to verify the results of feasibility and requirements analyses, much of the verification and validation of requirements and design is done by review. A review is a labor-intensive process. Users, designers, system engineers, and others scrutinize a specification for errors, usefulness, and other properties. Then, in a series of meetings, they discuss comments and objections to the specification. There is little automated support for reviews, and there have been few advances in the past 10 years in the way they have been conducted. Although many of the clerical aspects of such reviews are ripe for automation, the more difficult parts are likely to remain highly labor-intensive.

Coding

Coding activities generally consume about 10-20 percent of the effort in large-scale software development, but they have been more highly automated than any other part of the process. Perhaps the largest advance was the development of compilers that translate high-level languages into sequences of machine instructions. In addition, there is a continuing stream of new tools that help the coder to enter, edit, and debug his code.

[19] Boehm, *Software Engineering Economics*, op. cit., footnote 1, P. 40.

Advanced coding tools include editors, compilers, and debuggers that:

- incorporate syntactic knowledge of the language being used,
- allow the programmer to move freely between editing and debugging programs, and
- provide him with powerful means of browsing through the text of a program and analyzing the results of its execution.

Such tools generally reside either on a time-shared computer or on a workstation that is at the sole disposal of the programmer.

Current practice varies widely, from compilers used on batch machines (i.e., noninteractively, with little or no editing or debugging tools and with programmers relying principally on printouts for information), to state-of-the-art systems.

Showing Correctness and Utility of Code

Because code is the means of directing a computer's actions, it is the realization of the requirements and the implementation of the design. Although earlier stages in the development process might conceivably be reduced in scope and effort—or even eliminated-code to implement the system must still be written. To show that it is the accurate realization of the desired system, the code must be demonstrated to execute correctly and usefully. Technology to support such demonstrations has followed several different approaches.

The traditional approach is to test the software over a range of inputs that are deemed adequate to demonstrate correctness and usefulness. (The criteria for adequacy are generally determined by those responsible for accepting the software as adequate.) Testing technology is discussed in the next section.

Code reviews, similar to design and requirements reviews in structure, function, and labor-intensiveness, are also generally used during the coding process to find errors. As with other reviews, the nonclerical aspects of the process are unlikely to be automated.

Correctness proofs based on mathematical techniques are discussed in chapter 9 of this report. Although work in automating proofs of program correctness and finding and applying techniques that work for large programs started about 20 years ago, the technology is still inadequate for large, complex programs. There are no current signs of ideas that may lead to rapid progress.

Testing

Although there are different types of testing for different situations, the principles underlying different tests are the same: the program is executed using different sets of inputs and its behavior, particularly its output, is observed. Testing technology has advanced to the point where test inputs can often be automatically generated, test output can be automatically compared with desired output, and the parts of the program that have been executed during the test can be automatically identified. As with coding, the current practice varies widely. For simple, noncritical systems, none of the process maybe automated. For critical systems, considerable investment is often made in automating tests. For such systems, it is often important that test results be made visible and understandable to nontechnical users. As an example, elaborate computer-driven simulations are used in pre-flight testing of aircraft flight software. A pilot can test the behavior of the flight software without any knowledge of the code.

Integrated Support

Early tools designed to support software development or maintenance were aimed at solving specific problems, such as translating high-level languages to machine instructions, and were designed either to work alone or in cooperation with one or two other tools. Requirements and, particularly, design support tools did not interface well with coding and testing tools. More recent attempts at automating the software development and maintenance processes are aimed at developing *software engineering environments:* tools that are compatible with each other, that make it easy for the software engineer to switch his attention among different tasks in different phases of the software life cycle, and that support the entire software life cycle. Such environments are still in the development stage.

In recent years, efforts to provide automated management support have appeared. Such support might consist of providing an automated database containing information about the progress of a software development or maintenance project. Efficient tools for providing integrated management support should be appearing on the market shortly. One area that has enjoyed automated support for some time is change control, that is, keeping track of changes that have been proposed and made to

a system during its lifetime, long recognized as an important management need. Automated support systems designed just for change control have been available for at least 10 years.

Incremental Development

To avoid the problems associated with attempting to develop a large, complex system at one time, an incremental development technique is often used. Systematic approaches for incremental development have been described in the literature for more than 10 years; example variations are iterative enhancement, and program family development.[14] More recently, incremental development has been incorporated into a risk-based approach to development called the spiral approach.[15] In this approach, each developmental increment is accompanied by risk analyses. When deemed worth the risk, a complete development cycle, which maybe similar to the one described in the preceding, is used. Incremental development has been used by DoD in a variety of forms for a number of years, and should not be expected to result in a major improvement in software dependability or productivity.

Other Paradigms

The preceding discussion is oriented towards the standard DoD software life cycle. Other paradigms for the software life cycle have been suggested. Some expand on the life cycle, such as a recently proposed model by Boehm that incorporates risk-assessment and incremental development. Others attempt to eliminate or merge existing steps, such as object-oriented programming using languages like Smalltalk that lend themselves to rapid change. Some introduce new or improved technology to change the nature of existing steps, such as the Cleanroom method.

Object-Oriented Programming

Object-oriented programming is based on several different ideas that are used differently by advocates of the technique.

> The term object-oriented programming has been used to mean different things, but one thing these languages have in common is objects. Objects are entities that combine the properties of procedures and data since they perform computations and save local states.[16]

In many versions of the object-oriented model, the role of formal requirements and design specification is reduced in favor of quickly producing different versions of a program until one is attained that exhibits the desired behavior. Although this technique appears to work well on a small-scale, it has yet to be tried on large-scale programs that require the cooperation of many programmers and that are to be long-lived. Most likely, some of the ideas and tools that facilitate change in languages, like Smalltalk, will be incorporated into the software engineering environments under development for the standard DoD life cycle, where they will help make a modest improvement in productivity.

The Cleanroom Method

The Cleanroom method is an approach to software engineering recently developed at IBM.[17] The method requires programmers to verify their programs, using mathematically-based functional verification methods developed at IBM. Programmers are not permitted to debug or test their own programs; testing is done by a separate test group. Furthermore, the test process is based on statistical methods that permit statistically valid estimates of mean time to failure to be calculated from test results.[18] Reported Cleanroom experience includes three projects, the largest containing 45,000 lines of code. There is no reported accumulated operational experience with software developed using this technique. Proponents believe the technique will scale up to programs of the size and complexity needed for SDI.

Automatic Programming

In another suggested paradigm that would eliminate much of the requirements and design phases, programs would be automatically generated directly from a requirements specification language that might read like a mathematized version of English. This idea is not new; the term auto-

[14] V.R. Basili and A.J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering SE-1(4):390-396,* December 1975. See also D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering 5(2),* March 1979.

[15] B. Boehm, TRW Corp., *A Spiral Model of Software Development and Enhancement,* TRW technical report 21-371-85, 1985.

[16] Mark Stefik and Daniel G. Bobrow, "Object-Oriented programming: Themes and Variations," *The AI Magazine* 6(4):40-62, winter 1986.

[17] Harlan D. Mills, "Cleanroom Software Engineering," to be published in *IEEE Transactions on Software Engineering*

[18] P. Allen, Michael Dyer, and D. Harlan, "Certifying the Reliability of So ftware," *IEEE Transactions on Software Engineering, SE-12(l):3-11,* January 1986.

matic programming was applied in discussions of programming languages as early as 1948.[19] As programming languages became more powerful, the level of expectation for automatic programming rose. The technology to implement this paradigm in such a way that design specifications, as now used, would be unneeded, is still well beyond the state of the art.

Artificial Intelligence

Artificial intelligence (AI) is sometimes asserted to be a technology that would be needed to build the software for an SDI BMD system. In a critique of AI as applied to SDI, David Parnas points out two different definitions of AI that are currently used:

> AI-1: The use of computers to solve problems that previously could only be solved by applying human intelligence.
>
> AI-2: The use of a specific set of programming techniques known as heuristic or rule-based programming. In this approach, human experts are studied to determine what heuristics or rules of thumb they use in solving problems. Usually they are asked for their rules. These rules are then encoded as input to a program that attempts to behave in accordance with them. In other words the program is designed to solve a problem the way that humans seem to solve it.[20]

Much of the investment in AI technology today seems to be based on AI-2. The result is likely to be several systems that work well in limited applications where the rules for solving a problem are well-known, relatively few in number, and consistent with each other. For battle management and other complex SDI computing problems, such an approach is unlikely to apply: the rules for conducting a battle in space against an opponent, who may use unforeseen strategy or tactics, are not well known.

Since AI-1 may be considered as a set of problems, such as writing a computer program that can translate English to Russian, it cannot be truly characterized as having an underlying technology. Solutions to such problems may or may not use AI-2, or any other technology. Accordingly, the state of the art in AI-1 can only be considered on a problem-by-problem basis, and a technological assessment cannot be made. Since it lacks a unifying concept or technology, there is not much sense in talking about "applying' AI-1 to SDI battle management problems until a specific set of battle management 'problems and their solutions is specified.

Technology Summary

Much of the current software technology work may be viewed as consolidation: the development of tools to support existing methodologies. This view is especially true for DoD, whose recent software technology investments are aimed at providing automated support for software to be developed in the Ada language. Both within and without DoD, particular emphasis is being given to *software engineering environments:* tools that are compatible with each other, that make it easy for the software engineer to switch his attention among different tasks in different phases of the software life cycle, and that support the entire software life cycle. This emphasis is likely a result of a growing recognition by software engineers that although they have spent considerable time helping to automate other industries, they have been slow to automate the software development and maintenance industry.

The difficult problems of how to go about creating, analyzing, specifying, and validating requirements and design, and validating that implementations satisfy requirements, are still open research problems on which progress is slow.

Measuring Improvement

Because the quality of software depends so strongly on the quality of the software development process, both the process and the product need to be measured to understand where process improvements are needed and what their effect is. As previously noted, increases in product scale result in a shift in the factors determining success. Measurements made on small scale developments cannot be generalized to large scale developments. As a result, laboratory-style measurements are of little help in trying to determine the factors affecting the development of BMD software. To be useful, measurements must be made of the actual production process, with the attendant risks of affecting the process. Since data from such measurements gives considerable insight into the practices used by a company, it is considered by most companies to be sensitive and is rarely available for study outside of the company. As a result, there is little to chance to separate the effects of differ-

[19] For a discussion of automatic programming, See David L. Parnas, "Can Automatic Programming Solve The SDI Software Problem," in "Software Aspects of Strategic Defense Systems," *American Scientist*, September-October 1985, pp.432-40.

[20] "David L. Parnas, "Artificial Intelligence and the Strategic Defense Initiative," *ibid.*

ent factors by comparing data from different development environments. Outside of internal company studies, the few studies of software available from measurements in production environments come either from NASA's Software Engineering Laboratory, or from the Data and Analysis Center for Software, supported by the Rome Air Development Center.

Scaling up to the size estimated for the SDI battle management software means that new developmental problems will be encountered and that existing measurements will not apply well. Estimates for the size of the SDI battle management software range from 7 million lines of code to 60 million lines of code, depending on the estimator and system architecture. The largest operational systems today that could be said to be similar to BMD systems contain about **3-4** million lines of code.[21]

[21] The AEGIS software is in this category. The software for AT&T's 5ESS™ switching system, although not a good model for BMD software, is also in this size range. The SAFEGUARD system, not currently operational, was slightly smaller.