Perspectives on Software Engineering

MARVIN V. ZELKOWITZ

Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C 20234, and Department of Computer Science, University of Maryland, College Park, Maryland 20742

Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliability and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed Solutions from both the management and the programmer points of view are then given for many of these problem areas.

Keywords and Phrases: certification, chief programmer team, program correctness, program design language (PDL), software reliability, software development life cycle, software engineering, structured programming, top-down design, top-down development, validation, verification

CR Categories: 1.3, 4.0, 4.6

INTRODUCTION

Software development usually proceeds in one of two ways: either the programmer works alone in designing, implementing, and testing a software system, or he is a member of a group of from three up to several hundred, working together on a large software system. Although software engineering embraces both approaches, here we are interested mainly in large-scale program development.

When the Verrazano Narrows Bridge in New York City was started in 1959, officials estimated that it would cost $325 million and be completed by 1965. It is the largest suspension bridge ever built, yet it was completed in November 1964, on target and within budget [ENR61, ENR64]. No similar pattern has been observed when we build software systems larger than those which had been built previously.

Software is often delivered late. It is frequently unreliable and usually expensive to maintain. The IBM OS project, which involved over 5,000 man-years of effort, was years late [BROO75]. Why is bridge engineering so exact while software engineering flounders so?

Part of the answer lies in the greater ease with which a civil engineer can see the added complexity of a larger bridge than a software engineer the complexity of a larger program. Part of today's "software problem" stems from our attempt to extrapolate from personal experiences with smaller programs to large systems programming projects.

We begin here by outlining the general approach used in developing program products, emphasizing aspects which are still poorly understood. Later, we enumerate the techniques which have been used to solve these problems. We do not attempt to cover all of the relevant topics in depth, but we give many references for further reading.

Software engineers are currently study-

CONTENTS

INTRODUCTION
1 STAGES OF SOFTWARE DEVELOPMENT
Requirements Analysis
Specification
Design
Coding
Testing
Operation and Maintenance
Themes of Software Engineering
2 MANAGEMENT ISSUES
Size and Cost Control
Project Personnel
Estimation Techniques
Milestones
Development Tools
Reliability
Conceptual Integrity
Continual System Validation
3 PROGRAMMER ISSUES
Verification and Validation
Automated Tools
Certification
Formal Testing
Mean Time Between Failure
Error Days
Programming Techniques
Structured Programming
System Design
Performance Issues
Algorithm Analysis
Efficiency
Theory of Specifications
SUMMARY
ACKNOWLEDGMENTS
REFERENCES

ing the causes of these problems and the mechanisms of software development. They seek both constraints on programming which will render software less expensive and more reliable and also the theoretical foundations upon which programs are built. Software engineering is not the same as programming, although programming is an important component. It is not the study of compilers and operating systems, although compiler writers and operating system implementors use similar techniques. It is not electrical engineering, although electronics does provide the basis for implementing the computer [JEFF77].

Software engineering is interdisciplinary. It uses mathematics to analyze and certify algorithms, engineering to estimate costs and define tradeoffs, and management science to define requirements, assess risks, oversee personnel, and monitor progress.

1. STAGES OF SOFTWARE DEVELOPMENT

The complexity of a large software system surpasses the comprehension of any one individual. To better control the development of a project, software managers have identified six separate stages through which software projects pass; these stages are collectively called the *software development life cycle*:

- Requirements analysis;
- Specification;
- Design;
- Coding;
- Testing;
- Operation and maintenance.

Figure 1, a pie chart, shows the approximate amount of time each stage takes. The stages are discussed in the following subsections.

Requirements Analysis

This first stage, curiously absent from many projects, defines the requirements for an acceptable solution to the problem. The statement "Write a COBOL program of not more than 50,000 words to produce payroll checks" is not a requirement; it is the partial specification of a computer solution to the problem. The computer is merely a tool for solving the problem. The requirements analysis focuses on the interface between

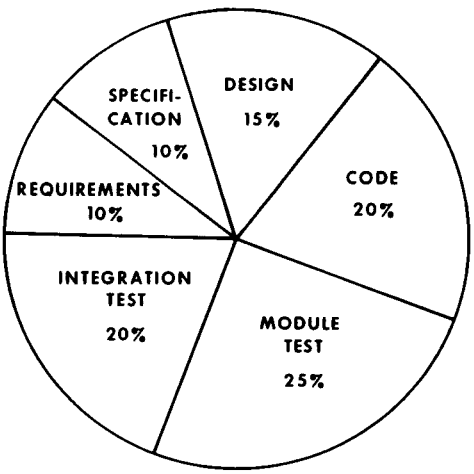

FIGURE 1. Effort required on various development activities (excluding maintenance)

the tool and the people who need to use it. For example, a company may consider several methods of paying its employees: 1) pay employees in cash; 2) use a computer to print payroll checks; 3) produce payroll checks manually; or 4) deposit payroll directly into employees' bank accounts.

Other aspects, such as processing time, costs, error probability, and chance of fraud or theft, must be considered among the basic requirements before an appropriate solution may be chosen. A requirements analysis can aid in understanding both the problem and the tradeoffs among conflicting constraints, thereby contributing to the best solution.

Hard requirements and the optional features must be distinguished. Are there time or space limitations? What facilities of the system are likely to change in the future? What facilities will be needed to maintain different versions of the system at different locations?

The resources needed to implement the system must be determined. How much money is available for the project? How much is actually needed? How many computers or computer services are affordable? What personnel are available? Can existing software be used? After the first questions are answered, project schedules must be planned. How will progress be controlled and monitored? What has been learned from previous efforts? What checkpoints will be inserted to measure this progress? Once all these questions have been answered, specification of a computer solution to the problem may begin.

Specification

While requirement analysis seeks to determine whether to use a computer, *specification* (also called *definition* [FIFE77]) seeks to define precisely what the computer is to do. What are the inputs and outputs? In the payroll example: Are employee records in a disk file? On tape? What is the format for each record in the file? What is the format for the output? Are checks to be printed? Is another tape to be written containing information for printing the checks offline? Will printed reports accompany the checks? What algorithms will be needed for computing deductions such as tax, unemployment and health insurance, or pension payments?

Since commercial systems process considerable amounts of data, the database is a central concern. What files are needed? How will they be formatted, accessed, updated, and deleted?

When the new system supersedes an older process (for example, when an automatic payroll system replaces a manual system), the conversion of the existing database to the new format must be part of the design. Conversion may require a special program which is discarded after its first and only use. Since the company may be using the older system in its day-to-day operation, bringing the new system online presents a problem. Can the old and the new systems run side by side for awhile?

The answers to these questions are set forth in the *functional specification*, a document describing the proposed computer solution. This document is important throughout the project. By defining the project, the specification gives both the purchaser and the developer a concrete description. The more precise the specifications are, the less likely will be errors, confusion, or recriminations later. The specifications enable test data to be developed early; this means that the performance of the system can be tested objectively, since the test data will not be influenced by implementation. Because it describes the scope of the solution, this document can be used for initial estimates of time, personnel, and other resources needed for the project.

These specifications define only what the system is to do, but not how to do it. Detailed algorithms for implementation are premature and may unduly constrain the designers.

Design

In the design stage, the algorithms called for in the specifications are developed, and the overall structure of the computer system takes shape. The system must be divided into small parts, each of which is the responsibility of an individual or a small

team. Each such module thus defined must have its constraints: its function, size, and speed.

As submodules are specified, they are represented in a tree diagram showing the nesting of the system's components. Figure 2 illustrates this for a typical compiler. This illustration, sometimes called a *baseline diagram*, is not by itself an adequate specification of the system.

Because the solution may not be known when the design stage starts, decomposition into small modules may be quite difficult. For older applications (such as compiler writing) this process may become standardized, but for new ones (such as defense systems or spacecraft control) it may be quite difficult.

A common problem is that the buyer of a system often does not know exactly what he wants, especially in state-of-the-art areas such as defense systems. As he sees the project evolve, the buyer often changes the specifications. If this occurs too often, the project may flounder. We discuss this problem later.

Coding

Coding is usually the easiest stage. High-level languages and structured programming simplify the task. In one study, Boehm [BOEH75] found that 64% of all errors occurred in design, but only 36% in coding. Hamilton and Zeldin [HAMI76] report that in the NASA Apollo project about 73% of all errors were design errors. We have mastered coding better than any other stage of software development.

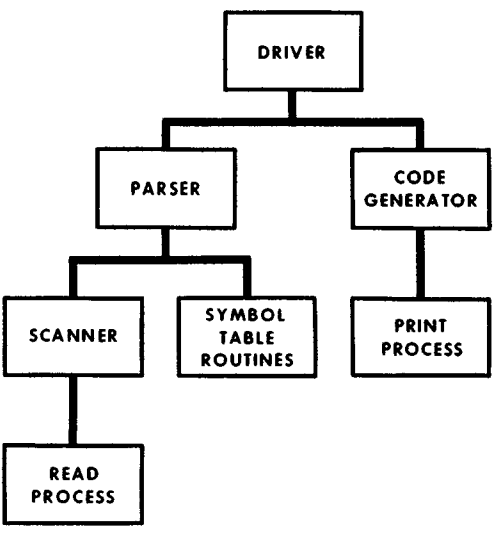

FIGURE 2. Sample baseline diagram for a compiler.

Testing

The testing stage may require up to half of the total effort. Inadequately planned testing often results in woefully late deliveries.

During testing the system is presented with data representative of that for the finished system; thus test data cannot be chosen at random. The test plan should, in fact, be designed early and most of the test data should be specified during the design stage of the project.

Testing is divided into three distinct operations:

1) *Module testing* subjects each module to the test data supplied by the programmer. A test driver simulates the software environment of the module by containing dummy routines to take the place of the actual subroutines that the tested module calls. Module testing is sometimes called *unit testing*. A module that passes these tests is released for integration testing.
2) *Integration testing* tests groups of components together. Eventually, this procedure produces a completely tested system. Integration testing frequently reveals errors missed in module tests. Correcting them may account for about a quarter of the total effort.
3) *Systems testing* involves the test of the completed system by an outside group. The independence of this group is important.

The buyer may also insist on his own systems test, or *acceptance test*, before formally accepting the product. Comparison of the performance of several systems (such as those of a given software product already available from several sources) is called *benchmark testing*.

During testing, many criteria are used to determine correct program execution. Among other important criteria, the pro-

gram is considered correct if:

1) every statement has been executed at least once by the test data;
2) every path through the program has been executed at least once by the test data; and
3) for each specification of the program, test data demonstrate that the program performs the particular specification correctly.

These three different criteria show that there is no single acceptable criterion defining a "well-tested" program. Goodenough and Gerhart [GOOD76] proposed a set of consistent definitions for "testing" and showed that some of these definitions of testing are, in theory, insufficient. We return to this subject later. For a survey of good testing techniques, see [HUAN75].

Closely related to testing are verification and validation (V/V). A system is *validated* when testing shows that the system performs according to its specifications. A system is *verified* when it has been proved to meet its specifications. Current technology is inadequate for achieving both these objectives. A validated system may misbehave for cases not included in the test data. A verified system is correct relative only to the initial specifications and assumptions about the operating environment; formal proofs tend to be lengthy, making them subject to error or incredulity. *Certification* sometimes refers to the overall process of creating a correct program by validation and verification.

In certifying a program, three terms must be distinguished. A *failure* in a system is an event which marks a violation of the system's specifications. An *error* is an item of information which, when processed by the normal algorithms of the system, produces a failure. Since error recovery may be built into the program (for example, ON units in PL/I), not every error will produce a failure. A *fault* is a mechanical or algorithmic defect which generates an error (for example, a programming "bug") [DENN76a].

Reliability is a concept which must not be confused with correctness. A *correct* program is one that has been proved to meet its specifications. In contrast, a *reliable* program need not be correct, but gives acceptable answers even if the data or environment do not meet the assumptions made about them. We would like a system to be highly robust, that is, to accept a large class of input data and to process it correctly under adverse conditions. Parnas [PARN75] describes a correct system as one that is free from faults and has no errors in its internal data. A program is reliable if failures do not seriously impair its satisfactory operation.

Operating systems with "fail-soft" procedures illustrate the difference between reliability and correctness. A detected error causes the system to shut down without losing information, possibly restarting after error recovery. Such a system may not be correct because it is subject to errors, but it is reliable because of its consistent operation. A real-time program may be correct as long as a sensor reports correctly, but it may be unreliable if bad sensor readings have not been considered.

Operation and Maintenance

Figure 1 shows the disposition of software costs in developing a new project. But this can be the wrong chart! The activities noted in Figure 1 are only 25% to 33% of the effort required during the life of the system. Figure 3 illustrates that maintenance costs ultimately dwarf development costs.

No computer system is immutable. Since a buyer seldom knows what he wants, he seldom is satisfied. Probably, he will request changes in the delivered system. Errors missed in testing will later be discovered. Different installations will need special modifications for local conditions. The management of multiple copies of a system is another difficult problem that must be handled early in development. Once the first line of code is written, the structure of the resulting maintenance operation may already be fixed, so it is best to plan for it then.

The division of effort indicated in Figure 3 greatly affects system development. Because of hidden maintenance costs, techniques that rush development and provide

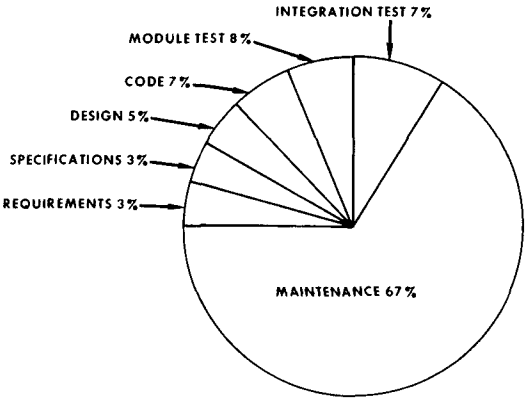

FIGURE 3. True effort on many large-scale software systems.

for very early initial implementation may be trading early execution for a much more extensive maintenance operation.

The maintenance problem is sometimes referred to as the "parts number explosion." For example, a certain system contains components A, B, and C. Installation I finds and reports an error. The developer fixes the error and sends a corrected module A′ to all installations using the system.

Installations II and III ignore the replacement and continue with the original system. Installations I and II discover another error in module A. The developer must now determine whether both of these errors are the same, since different versions of module A are involved. The correction of this error involves correction of both A′ (for I) and A (for II) yielding A″ and A‴. There are now three versions of the system.

To avoid this growth, systems often receive updates, called releases, at fixed intervals. A useful tool for dealing with myriad maintenance problems is a "systems database" started during the specifications stage. This database records the characteristics of the different installations. It includes the procedures for reporting, testing, and repairing errors before distributing the corrections.

Themes of Software Engineering

It should be clear that each software development stage may influence earlier stages. The writing of specifications gives feedback for evaluating resource requirements; the design often reveals flaws in these specifications; coding, testing, and operation reveal problems in design. The goals of software engineering are thus to:

- Use techniques that manage system complexity.
- Increase system reliability and correctness.
- Develop techniques to predict software costs more accurately.

In the following sections, we discuss approaches to some of these problems. The list of techniques is divided into management and programmer issues. Management issues concern the effective organization of personnel on a project. Programmer issues concern the techniques used by individual programmers to improve their performance.

2. MANAGEMENT ISSUES

A manager controls two major resources: personnel and computer equipment. This section surveys techniques for optimizing the use of these resources.

Size and Cost Control

A project may fail when management is not aware of developing problems; a year's delay comes "one day at a time" [BROO75]. Faced with catastrophic failure (for example, needed hardware is delayed six months), a resourceful manager can usually find alternatives. However, it is easy to ignore day-to-day problems (such as sick employees or many errors during testing).

Most problems occur at the interfaces of modules written by different programmers. Since the number of such interfaces is on the order of the square of the number of individuals involved, the problem becomes unwieldy when the number of persons in a development group grows to four or more.

As an example of the communications problem, assume that a single programmer is capable of writing a 5,000-line program in a year, and that a programming system requires about 50,000 lines of code and is to be completed in two years. Five programmers would seem to be sufficient (see Figure 4a).

However, the five programmers must communicate with one another. Such communication takes time and also causes some loss in productivity since finding misunderstood aspects will require additional testing. For this simple analysis, assume that each communication path "costs" a programmer 250 lines of code per year. Each of the five programmers, therefore, can produce only 4,000 lines per year and only 40,000 lines are completed within two years (see Figure 4b).

This means that eight programmers producing 3,250 lines per year are actually needed in order to produce the required 50,000. A manager is required for direction of this large effort. Therefore, in summary, eight programmers and a manager, each producing an average of 3,000 lines per year, are actually needed (see Figure 4c).

As we shall see, simply counting lines of code is not a good way to estimate productivity. The figures in this example are only given to illustrate a point, but they are representative of the problem. There are also techniques designed to limit this communications "explosion" and to increase programmer productivity.

Project Personnel

Software can usually be divided into three categories: 1) control programs (such as operating systems), 2) systems programs (such as compilers), and 3) applications programs (such as file management systems). A single programmer working on a control program can produce about 600 lines of code per year, whereas he can produce about 2,000 lines if working on a systems program and about 6,000 if working on an applications program [WOLV74]. The type of task certainly affects the productivity that can be expected from a given programmer. However, as the previous example demonstrates, the organization of personnel also affects performance. For example, with the approach of deadlines, docu-

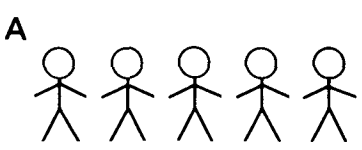

FIGURE 4(a) Single projects. 5,000 lines per year = 50,000 lines in two years (no communication between programmers)

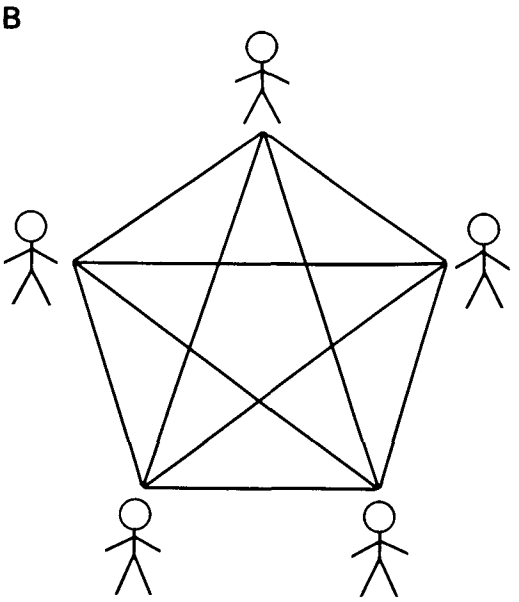

FIGURE 4(b). Five-member group· 4,000 lines per year = 40,000 lines in two years (ten communication pairs).

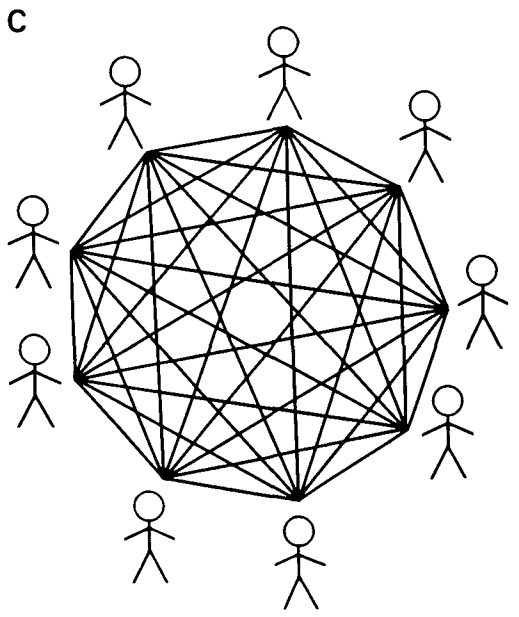

FIGURE 4(c). Nine-member team: 3,000 lines per year = 50,000 lines in two years (36 communication pairs).

mentation is often given lower priority. However, since 70% of the total system cost may occur during the maintenance state (where the documentation is heavily used), this may be a false economy of effort.

Use of a librarian is one way to avoid this problem. A librarian provides the interface between the programmer and the computer. Programs are coded and given to the librarian for insertion into the online project library. The actual debugging of the module is carried out by the programmer, but changes to the official module in the library are made by the librarian. The use of a library is further enhanced when an online data management system is used.

The use of a librarian has another beneficial effect. All changes in modules in the project library are handled by one individual and are easy to monitor; they are often reviewed by the project manager before insertion. This prevents "midnight patches" from being quickly incorporated into a system and forces the programmer to think carefully about each change. It also gives the manager disciplined product control and helps with audit trails.

On larger projects, a technical writer may perform much of the documentation, thus freeing programmers for the tasks for which they are most skilled.

The culmination of this trend is the *chief programmer team* concept developed by IBM [BAKE72]. The concept recognizes that programmers have different levels of competence; therefore, the most competent should do the major work, while others function in supporting roles. As the earlier example shows, interfacing problems greatly reduce programmer productivity. The chief programmer team is one way of limiting this complexity.

The chief programmer, an excellent programmer and a creative and well-disciplined individual, is the head of the team. He may be five or more times more productive than the lowest member of the team [BOEH77]. He functions as the technical manager of the project, designs the system, and writes the top-level interfaces for all major modules.

If a project is large, a team may also have an administrative manager to handle such responsibilities as budgeting time, vacations, office space, and other resources, and reporting to upper-level management. The administrative manager often administers several programming teams.

The backup programmer works with the chief programmer and fills in details assigned by the chief programmer. Should the chief programmer leave the project, the backup programmer would take over. This means that he also must be an excellent programmer. The backup programmer also fulfills an important role by providing the chief programmer with a peer with whom he can discuss the design.

There are also two or three junior programmers assigned to the team to write the low-level modules defined by the chief programmer. The term "junior" in this context means "less experienced," not "less capable." As Boehm states, the best results occur with fewer and better people.

Using the example illustrated by Figure 4, a chief programmer team of five individuals has only seven communications paths, and the chief programmer, being that rare individual, can produce more than his quota of 5,000 lines (see Figure 5). Thus productivity per programmer could be greater than 5,000 lines per year, instead of the previous figure of only 4,000.

The team has a librarian to manage the project library—both the online module library and the offline project documentation (also called the project notebook). The project notebook contains, among other things, records of compilations and test runs of all modules. It is important to the team structure, since all development is now accountable and open for inspection, and code is no longer the "private property" of any individual programmer.

Programmers have traditionally been reluctant to exhibit their products until completion, since discovered errors have traditionally been viewed as a personal failure. The absurdity of this approach is clear enough. If the ego element is removed from programming, programmers may openly ask others for advice when they need it, instead of trying to solve all problems themselves [WEIN71].

The team may include other supporting

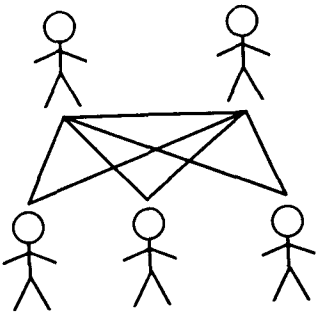

FIGURE 5. Fewer communications paths in a chief programmer team.

personnel such as secretaries and technical writers. Experience shows that ten is the upper bound to team size.

This structure, however, will not solve all problems in development. With a smaller number of individuals involved, competence is crucial. It is not possible to "work around" a nonproductive individual as one might do in a large project. There are also extremely large projects where a group of ten is simply too small to tackle development. Larger teams are not efficient.

A man-month, or the amount of work performed by one individual in one month, is a deceptive measure for estimating project productivity. A project requiring four programmers for a year cannot be completed by 48 programmers in one month. The example of the 50,000 line system needed in two years shows some of the problems inherent in trying to exchange programmers for time. "Adding manpower to a late software project makes it later" [BROO75]. New personnel divert existing personnel needed to train them; they require more supervision; they complicate communication and interfere with the design since they are unfamiliar with the project structure.

However, man-months do serve a purpose as a useful measure of project costs. By adding more data, such as the rate of using man-months, accurate cost estimation techniques can be utilized. These are explained in the following subsection.

Estimation Techniques

One of the most important aspects of engineering is estimating the resources needed to complete a project. As previously mentioned, the Verrazano Narrows Bridge in New York City was completed at the projected time and within the estimated budget. How was such accuracy achieved?

Most engineering disciplines have highly developed methods of estimating resource needs. One such technique is the following [GALL65]:

1) Develop an outline of the requirements from the Request for Quotation (RFQ);
2) Gather similar information, for example, data from similar projects;
3) Select the basic relevant data;
4) Develop estimates;
5) Make the final evaluation.

Although this approach has been advocated for software development, software projects have difficulty passing Step 1 [WOLV74]. Engineers have been building bridges for 6,000 years but software systems for only 30 years. Prior experience to develop the true requirements may not be available. Moreover, with very little background to build on, the developer has little knowledge of similar systems to use in evaluation (Step 2).

In developing the estimates (Step 4), the following tasks must be undertaken:

4a) Compare the project to similar previous projects.
4b) Divide the project into units and compare each unit with similar units;
4c) Schedule work and estimate resources by the month.
4d) Develop standards that can be applied to work.

Note that for Step 4a), the lack of previous experience presents a continuing problem. Also, for Step 4d), an adequate set of standards does not yet exist.

Experience is the key to accurate estimation. Even civil engineering projects may fail badly when established techniques are not followed. Although the Verrazano Narrows Bridge was the world's largest suspension bridge, its engineers had much experience with other similar structures. On the other hand, the Alaskan oil pipeline was estimated to cost $900 million, yet by mid-

1977 the cost had risen past $9 billion [ENR77]. In this case, the design was altered continuously as the federal government imposed new environmental standards (that is, changing specifications), and new technologies were needed to move large quantities of oil in a cold weather environment. Previous experience was only marginally helpful.

Results from computer hardware reliability theory are now starting to play a role in software estimation [PUTN77]. The cumulative expenditures over time for large-scale projects have been found to agree closely with the following equations:

$$E = K(1-e^{-at^2})$$

where E is the total amount spent on the project up to time t, K is the total cost of the project, and a is a measure of the maximum expenditures for any one time period. This relationship is usually expressed in its differential form, called a Rayleigh curve:

$$E' = 2Kate^{-at^2}$$

where E' is the rate of expenditures, or the amount spent on the project during year number t. Since 70% of the cost of a project occurs during the maintenance stage, it is not surprising that the maximum expenditures will occur just before the product is released, a time when it is usually assumed that the effort is winding down before termination (see Figure 6).

The Rayleigh curve has two parameters, K and a; however, a system can be described by three general characteristics: 1) total cost, 2) rate of expenditure, and 3) completion date. Two of these characteristics are enough to determine the constants K and a. When a project is initiated, the proposed budget is an estimate of K, and the available personnel permits a to be calculated. Assuming that requirement analysis determines that these figures represent an accurate assessment of the complexity of the problem, the estimated completion date (the date when the expenditures reach a maximum) can be computed, and thus cannot be set arbitrarily during the requirements or specification stage. This method provides the basis for a cost estimation strategy that has been applied to smaller projects in the 100 man-month range [BASI78]. We may be close to a mathematical theory of cost estimation which will greatly reduce our need to "guess" at project costs.

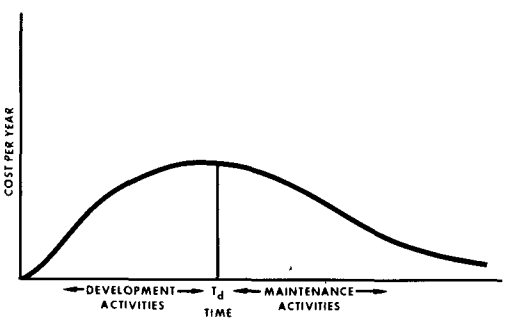

FIGURE 6. Yearly rate of expenditures approximates the Rayleigh curve. Total cost (area under curve) $= K$, $a = 1/T_d^2$, rate $= 2Kate^{-at^2}$

Milestones

A *milestone* is the specification of a demonstrable event in the development of a project. Milestones are scheduled by management to measure progress. "Coding is 90% complete" is not a milestone because the manager cannot know when 90% of the code is complete until the project itself is complete.

There are many candidates for milestones: publication of the functional specifications, writing of individual module designs, module compiling without errors, units that have been tested successfully, and so on. Milestones are scheduled fairly often to detect early slippage. PERT charts may be used to estimate the effects of slippage in one stage on later stages.

Reporting forms can give information useful for estimating when a future milestone will be reached. A general project summary, describing such overall characteristics as system size, cost, completion dates, or complexity, can be resubmitted with each milestone. Change reports can be submitted each time a module is altered. The use of a librarian probably means that such a form already exists. Weekly personnel and computer reports monitor expenditures. Although they add a minor overhead to the project, the information helps management keep abreast of progress [BASI78, WALS77].

Development Tools

Compilers and certain debugging facilities have been available for some time. In contrast, other programming aids are new and experience with them is less estensive. Cross referencing, attribute listings, and symbolic storage maps are examples of such aids. Auditors or database systems can help to control the organization of the developing system. The Problem Statement Language/Problem Statement Analyzer (PSL/PSA) of the ISDOS project of the University of Michigan is one of the first database systems for providing a module library for storing source code, and includes a language for specifying interfaces in system design which can be checked automatically [TEIC77]. RSL/SSL is a similar system designed to specify requirements and to design interfaces via a data management system [DAVI77].

An alternative approach is the Programmer's Workbench developed by Bell Telephone Laboratories [DOLO76]. A PDP 11 based system provides a set of support routines for module development, library maintenance, documentation, and testing. Proper use of these facilities allows accessing information in an easier, controlled environment.

Reliability

Conceptual Integrity

Conceptual integrity, uniformity of style and simplicity of structure, are usually achieved by minimizing the number of individuals in the project. A chief programmer team greatly enhances conceptual integrity.

A small group minimizes contradictory aspects of a design. In the PL/I language, for example, the PICTURE attribute declaration may be abbreviated as either PIC or P, but in format specifications it may only be P [ANSI76]. In FORTRAN, the right side of an assignment statement can be an arbitrary arithmetic expression, but DO loop indices must be integer constants or variables, and subscripts to arrays are limited to seven basic forms [ANSI66]. These are difficult idiosyncracies to remember. They illustrate a lack of conceptual integrity that can arise when many people with different objectives become involved in a project. A consistent design is less prone to errors because the user can follow a simple set of rules.

Continual System Validation

A *walkthrough* is a management review to discover errors in a system. In one study, TRW discovered that the cost of fixing an error at the coding stage is about twice that of fixing it at the design stage, and catching it in testing costs about ten times as much as it does in design [BOEH76].

A walkthrough is scheduled periodically for all personnel. In attendance are the project manager (chief programmer), the person reviewed, and several others knowledgeable about the project. One section of the system is selected for review and each individual is given information about that section (for example, design document for a design walkthrough, code for a coding walkthrough) before the review. The person being reviewed then describes the module under study.

The walkthrough is intended to detect errors, not to correct them. Also, the walkthrough is brief—not more than two hours. By explaining the design to others, the person reviewed is likely to discover vague specifications or missing conditions.

An important point for management is that the walkthrough is *not* for personnel evaluation. If the person reviewed perceives that he is being evaluated, he may attempt to cover up problems or present a rosy picture.

An informal yet very effective version of the walkthrough is *code reading*. A second programmer reviews the code for each module. This technique frequently turns up errors when the second reader, failing to understand some aspects of the code, asks the author for an explanation.

3. PROGRAMMER ISSUES

Each stage of the software development life cycle has its own set of problems and solutions. The most advanced techniques apply to the last stages; the first stages are the least developed. For example, testing and

debugging problems are apparent to every programmer; these tools are the oldest and most advanced. Techniques for improving coding were developed next. The most recent developments have related to requirements and specifications. Although many technical problems have not been solved, an effective methodology is emerging. Some of these techniques are presented in the following subsections.

Verification and Validation

Verification and validation (module and integration testing) of a system occupy about half of the development time of a project. Many debugging aids have been developed to facilitate this effort; most are implemented as programs to test some feature of a system.

Automated Tools

The earliest and most primitive debugging tools were the dump and the trace. A *dump* is a listing of the contents of the machine's memory. This listing can often reveal unintelligible data or errors. Unfortunately, a dump may not be taken until long "after the fact" and the cause of the error may not then be apparent. A *trace* is a printout showing the values of selected variables after each statement is executed. It may help a programmer to discover errors.

These techniques are not usually very effective because they supply much data with little or no interpretation. More advanced methods are needed to reduce this data to an intelligible form.

Flowgraph analyzers are capable of detecting references to variables which are never initialized or never reused after receiving a value; these usually indicate errors. Test data generators are also available. Assertion checkers validate that given conditions are true at indicated points of a program. Automatic verification systems have been implemented for small languages [KING69] and symbolic execution has been proposed as a practical means for validating programs in a more complex language. The PSL/PSA system is an example of a tool for assisting in design and specification. Symbolic dumps and traces are generated with compilers like PL/C [CONW73] or PLUM [ZELK75]. Ramamoorthy and Ho [RAMA75] survey many of these tools.

Certification

Programs can be verified at several levels. Conway [CONW78] lists eight different verification conditions:

- A program contains no syntactic errors.
- A program contains no compilation errors or faults during program execution.
- There exist test data for which the program gives correct answers.
- For typical sets of test data, the program gives correct answers.
- For difficult sets of test data, the program gives correct answers.
- For all possible sets of data which are valid with respect to the problem specification, the program gives correct answers.
- For all possible sets of valid test data and all likely conditions of erroneous input, the program gives correct answers.
- For all possible input, the program gives correct answers.

Some people are optimistic that one day complete automatic program verification will be possible. Today's tools operate a posteriori, demonstrating that a given program works. Tomorrow's tools will also operate a priori, helping to develop programs which are correct before they are ever run. Such tools can reduce the amount of testing required for a completed project [DIJK76].

Verification techniques have the following general structures. A program is represented by a flowchart. Associated with each arc in the flowchart is a predicate, called an *assertion*. If A_i is the assertion associated with an arc entering statement S, and A_j is the assertion on the arc following the statement, then the statement "If A_i is true, and if statement S is executed, then assertion A_j will be true" must be proved (see Figure 7).

This process can be repeated for each statement in a program. If A_1 is the assertion immediately preceding the input node to the flowchart (that is, the initial asser-

FIGURE 7. Assertions A_i and A_j surround each statement of a program.

tion), and if A_n is the assertion at the exit node (for example, the final assertion), then the statement "If A_1 is true, and the program is executed, then A_n is true" will be the theorem that states that the program meets its specifications (A_1 and A_n) (see Figure 8). This approach was formalized by Hoare [HOAR69] who defined a set of axioms for determining the effects upon the assertions (preconditions and postconditions) by each statement type in a language. Thus verifying program correctness reduces to proving a theorem of the predicate calculus.

Certification technique development is still in a preliminary stage and does not meet the challenge of a modern large system. In addition, axiomatic certification is weak in the sense that the output assertion is proved true only if the program terminates. Axiomatic methods are incapable of proving termination. However, termination can often be proved informally by the programmer.

A typical approach to proving that program loops terminate is the following:

1) Find some number P that is always nonnegative within the loop.
2) Show that for each execution of the loop, P is decremented by at least a fixed amount.

If both conditions are always true, the loop must terminate before P becomes negative. A programmer who uses such rules, even informally, will seldom write nonterminating loops.

Consider this program fragment:

```
while x<y do
    x := x+1
end
```

Let quantity P be the expression $y-x$, and let $P(i)$ refer to the value of P during the ith execution of the loop. Because $x<y$ must be true for each next iteration, $y-x$ is always nonnegative and condition 1) is satisfied for each execution of the loop. Since the loop contains the statement $x := x+1$, $P(i+1) = P(i)-1$, satisfying condition 2). Therefore the loop must terminate.

Certification will not solve all our software problems, although it is an important tool. Gerhart and Yelowitz [GERH76] have shown that there are many published "certified" programs that contain errors. Even experts err.

Formal Testing

Goodenough and Gerhart [GOOD75] have clarified the concepts of testing. A *domain* is the set of permissible inputs to a program, and a *test* is a subset of the domain. A *testing criterion* specifies what is to be tested (for example, specifications, all statements, all paths).

A test is *complete* if the test meets all the requirements of the testing criterion, and a complete test is *successful* if the program gives correct results for each input in the test.

With these definitions, we can define program reliability and validity. A program is *reliable* if every found error is revealed by every complete test. A program is *valid* if every error is revealed by some complete test.

With these definitions, several important results can be proved. Among these are:

- If a program is both reliable and valid, then it is correct if and only if any complete test is also successful.
- The criterion "execute every path" is not valid; there exist programs all of whose test sets succeed, but which produce the wrong results for some input.

While this framework is somewhat technical and is not applicable to all programming, it is an important step in formalizing this area. We now have a basis for talking

FIGURE 8. Predicates A_1 and A_n specify input-output behavior of a program.

about such concepts as reliability and correctness.

Mean Time Between Failure

While useful for focusing our attention, analogies with other engineering fields must be used with care. Reliability is one area of incomplete analogies. The concept of *mean time between failure (MTBF)* does not apply directly to software although it sometimes is used as if it does.

Systems built from physical components wear out; transistors fail; motors burn out; soldered joints break. This is also true for the hardware of the computer. However, the logical components of software are durable. A given program will always produce the same answer for the same input, as long as the hardware does not fail. When a software module "fails," it has been presented with an input that finally revealed an error present from the start.

The MTBF measures the time between revelations of errors. This, in turn, depends on the kinds of inputs presented. A compiler used only for short jobs from students may have a long MTBF; but if it is suddenly used for other applications, its MTBF may decrease sharply as unsuspected errors are exercised. A large MTBF can thus be interpreted only as an indication of possible reliability, not as a proof of it.

Error Days

Since formal certification of large classes of programs is still unattainable, techniques for estimating the validity of programs are still being considered. Most of these techniques measure the number of errors discovered, which are assumed to be representative of the total number of errors present in the system, and hence a measure of the reliability of the system.

Mills [MILL76] defines an *error day* as a measure stating that one error remains undetected in a system for one day. The total number of error days in a system is computed by summing, for each error, the length of time that error was in the system. A high error day count may reveal many errors (poor design) or long-lived errors (poor development).

The assumption is made that if a program is delivered with a low error day count, then there is a good chance that it will remain low during future use. However, two major problems remain before this measure can be widely used. First, it is difficult to discover when a particular error first entered a system. Second, it may be difficult to obtain such information from the developer of a delivered product.

Programming Techniques

Several authors have mentioned that the number of lines of code produced by a programmer in a given time tends to be independent of the language used. This implies that higher level languages enhance productivity [BROO75, HALS77]. This is true even though assembly language programs are potentially more efficient; their potential is seldom realized in practice.

The goals in developing early higher level languages were to be able to express clearly an algorithm and translate it into efficient machine language programs. The efficiency of the resulting code was all important. This led to some anomalies in FORTRAN arising from the structure of the IBM 704 for which it was developed (for example, the three-way branch of the arithmetic IF). ALGOL, which was developed as a machine-independent way of expressing algorithms, contained concepts whose implementation on conventional hardware was inefficient (e.g., recursion, call-by-name); this may explain why ALGOL is not widely used.

By the late 1960s it was accepted that the language should facilitate writing the program and that the machine should be designed to create an efficient run-time environment. Today there is a definite shift toward using the language to make programming and documentation easier and to produce reliable and correct software.

This does not mean, however, that efficiency is ignored today. Whereas PL/I permits the writing of simple programs whose execution time is quite long, PASCAL was designed to exclude constructs whose machine code is inefficient. Since hardware is less expensive than programmers, reliability has become a major factor: The pro-

grammer's task is made easier when the computer does more work.

Structured Programming

A major development in facilitating the programming task is known as *structured programming*, which has been erroneously called "**goto**less" programming. Fortunately, the debate about "to **goto** or not to **goto**" has mostly disappeared, and some clear ideas have emerged. The premise of structured programming is to use a small set of simple control and data structures with simple proof rules. A program then is built by nesting these statements inside each other. This method restricts the number of connections between program parts and thereby improves the comprehensibility and reliability of the program.

The **if-then-else**, **while-do**, and **sequence** statements are a commonly suggested set of control structures for this type of programming; however, there is nothing sacred about them. Knuth [KNUT74] has argued that the **goto** statement is irrelevant to the true goals of structured programming.

These simple control structures help programmers certify programs, even at an informal level. For example, a program can be represented as a function from its input data to its output data. Suppose $f(x)$ represents a segment of a program given by the following **if-then-else** statement:

$$\textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } h(x).$$

Because functions g and h are simpler than function f, their specifications should be simpler. If their specifications are known, the overall function f is defined by

$$f(x) = (p(x) \rightarrow g(x)) \vee (\neg p(x) \rightarrow h(x)).$$

The programmer can express the formal definition of f in terms of the simpler definitions of g and h.

Languages such as ALGOL, PASCAL, and certain subsets of PL/I contribute to good programming practices by providing these facilities. In order to repair FORTRAN's lack of structure, over 50 preprocessors for translating well-structured pseudo-FORTRAN programs into true FORTRAN have been developed [REIF76]. An **if-then-else** has been added to the new FORTRAN-77 standard, although a general **while** is still missing from the language.

System Design

A technique related to structured programming is *top-down design*, in which a programmer first formulates a subroutine as a single statement, which is then expanded into one or two of the basic control structures mentioned earlier. At each level the function is expanded in increasingly greater detail until the resulting description becomes the actual source language program in some programming language.

Using this approach, also called *stepwise refinement* [WIRT71, WIRT74], the program is hierarchically structured and is described by successive refinements. Each refinement is interpreted by referring to other refinements of which it is a component. Concerning this method, Wirth states:

> I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "topdown" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness The emerging modularity is particularly welcome if programs have to be adjusted to changed or extended specifications. [WIRT74, p. 251]

Operating systems are often modeled as hierarchies of *abstract* or *virtual* machines [BRIN77]. At the lowest level of the system is the physical hardware. Each new level provides additional *capabilities*, or allowable functions on data, and hides some of the details of a lower level. For example, if one level accesses the paging hardware of the computer and provides a large virtual memory for all other processes, other abstract machines at higher levels can be implemented as if they had unlimited memory since this detail is controlled by a lower level.

The concept of a *program design language* (PDL) to aid in this development

has been defined [CAIN75]. This type of language contains two structures: "outer" syntax of basic statement types, such as **if-then-else**, **while**, and **sequence** for connecting components, and an "inner" syntax that corresponds to the application being designed. The inner syntax is English statement oriented, and is expanded, step by step, until it expresses the algorithm in some programming language. Figure 9 represents an example of a PDL design.

It should be noted here that PSL/PSA and PDL complement each other. PSL/PSA is a specifications tool that validates correct data usage between two modules (interfaces). A system like PDL is useful for describing a given module at any level of detail. Both PSL/PSA and PDL can contribute to success in a large project.

Even though designed from the top down, many systems are implemented from the bottom up. Low-level routines are first coded with drivers to test them; then new modules, using these low-level routines, are added, and the system is built up.

Top-down development is another technique for implementing hierarchically structured programs. Here the top-level routines are written first and lower level routines, called *stubs*, are written to interface with these. The stubs return control after printing a simple message and may return some fixed sample test values. The stub is eventually replaced by the full module which now includes calls to other stubs. In this manner an entire system can be gradually developed.

If used carefully, this technique can be valuable; however, the system's correctness is assumed, not proved, until the last stub has been replaced [DENN76a]. The documentation specifies the assumptions on each stub. For example, if

$$f(x) = \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } h(x)$$

is a program fragment calling stubs g and h, then f will be correct only if the modules eventually replacing the stubs g and h are correct.

Via top-down development, a user sees the top-level interfaces in the system very early. He can then make changes relatively easily and soon. Another approach with the same goal is *iterative enhancement* [BASI 75]. Using this technique, a subset of the problem is first designed and implemented. This gives the user a running system early in the life cycle when changes are easier to make. This process is repeated to develop successively larger subsets until the final product is delivered.

Brooks [BROO75] believes that the first version of a system is always "thrown away," because the concrete specifications for a system are often not defined until the system is completed, a time when the initial product meets those specifications rather poorly. It is often cheaper and faster to rebuild a system from scratch than to try to modify an existing product to meet these specifications. However, a developer will often deliver such a modified system as a "pre-release" if a deadline is near and the purchaser is demanding results. The buyer then suffers with this version, replete with errors, until he throws it away or has the product rebuilt. Iterative enhancement can make rebuilding less chaotic since there is a running system (not meeting all the requirements) early in the development cycle.

```
max. PROCEDURE (list);
     /* Find maximum element in a list */
     DECLARE (maximum, next) integer,
     DECLARE list list of integers,
     maximum = first element of list,
     DO WHILE (more elements in list);
       next = next element of list,
       maximum = largest of next and maximum;
       END·
     RETURN (maximum);
     END max,
```

FIGURE 9. PDL of a program to find the largest element in a list (outer syntax is in upper case, inner syntax in lower case).

Performance Issues

The chosen algorithms and data structures have a much greater influence on program performance than code optimization or the programming language. Before choosing an algorithm, the programmer faces these questions:

- Can previously written software be used?
- If a new module must be written, what algorithms and data structures will give an efficient solution?

Programming languages usually include standard mathematical functions such as sine, logarithm, and square root. They give the programmer ready access to libraries of standard software packages. This allows the programmer to use results of previous work. In preparing programs for standard libraries, analysts have included many options in a single package. The effect can be a large cumbersome package which is inefficient because only a small part of it is applicable at any one time. This can be avoided by installing multiple versions of the module for each special case.

Many opportunities remain for more packaging and use of existing software. Difficulties in achieving this include:

- Identifying which standard algorithm to package. This is easier in mathematical areas such as statistical testing, integration, differentiation, and matrix computations than in many non-numerical areas such as business applications.
- Transporting and interfacing with packaged software. Some progress has been made with programs stored in read-only memories which plug into microprocessors, or with interface processors on computer networks. A major problem area lies in interfacing software directly to other software, since there are no conventions. Some help is afforded by such concepts as the "pipeline" in UNIX, which provides a general communications channel between programs [RITC74].

Algorithm Analysis

Sometimes the program specification is not changeable, and the analyst must find the best possible algorithm. Sometimes, however, the specifications can be altered to permit a more efficient solution. In some instances we can show that there are no algorithms guaranteed to be efficient in all cases; here approximate algorithms that are efficient in most cases but need not give exact solutions must be used.

The fast Fourier transform illustrates the most efficient form for computing the Fourier transform, a technique useful in waveform analysis [COOL65]. This transform is based on a finite set of points rather than on a complex integral which is harder to compute. Language analysis (parsing) in a compiler illustrates how changing the specification can permit a more efficient solution. Any string of N symbols in an arbitrary context-free language can be parsed in time of order $O(N**3)$ [YOUN67]; however, a programming language need not include all features of an arbitrary context-free language. PASCAL is an example of a language which can be parsed by a deterministic top-down parser in average time of order $O(N)$ [AHO72]. If we are free to set language specifications, we can choose the language and be rewarded with efficient compilers.

Many practical problems, such as job scheduling or network commodity flow, involve enumeration of a combinatorially large number of alternatives and selection of a best solution. In these cases it may be better to restrict the search for a suboptimal but good answer. We recommend the paper by Weide [WEID77] for a discussion of the issues and a state-of-the-art survey of algorithm analysis.

Efficiency

In many cases the results of algorithmic analysis are not extensive enough to help the programmer; thus we need to offer techniques which can help locate and remove sources of inefficiency. One such tool is an optimizing compiler which, for some languages, can yield significant improvements [LOWR69]. The value of such tools, however, is limited [KNUT71] and may be realized only for programs which are used often enough to justify the investment in optimization.

One of the most powerful aids is the *frequency histogram,* which reveals how often each statement of a program is executed. It is not unusual to find that 10% of the statements account for 80% of the execution time [KNUT71]. A programmer who concentrates on these "bottlenecks" in his algorithms can realize significant performance improvements at a minimum investment. This technique has been used in some interactive operating systems, such as

UNIX and MULTICS, which started out as high-level language operating systems. Bottlenecks have been replaced by assembly language routines in less than 20% of the system.

Theory of Specifications

One area of software engineering that is now under study is system specifications. The objective is to state the specifications early using a metalanguage. This places restrictions on the design and may help establish whether the specifications are met.

An early example of such a specification was the so-called "**goto**less programming" [DIJK68, KNUT74]. It is properly called "structured programming." It restricts the form of statements a programmer may use, but this restriction contributes to comprehensibility and enhances a correctness proof.

A second set of such rules employs the concepts of levels of abstraction, information hiding, and module interfacing to restrict access to the internal structure of data. Parnas [PARN72] formalized these ideas which were standard practices of expert programmers. He defines data as a collection of logical objects, each with a set of allowable states. Procedures can then be written to hide the representation of these objects inside separate modules. The user manipulates the objects by calling the special procedures.

Several languages that facilitate the use of these concepts have been developed. Among these are EUCLID [POPE77], CLU [LISK77], and ALPHARD [WULF76]. These languages permit programmers to define abstract data types having the property to encapsulate the representation of the logical objects [LISK75]. When concurrency is an issue, the use of abstract objects must be controlled by synchronization (for example, locks, signals); in this case the abstract type managers are called *monitors*.

Another kind of specification consists of "higher order software axioms" (HOS) [HAMI76], which are a set of six axioms that specify allowable interactions among processes in a real-time system. One axiom prohibits a process from controlling its own execution, thereby ruling out recursion in a design. Another axiom states that no module controls its own input data space and is therefore unable to alter its input variables. While these axioms are not complete, they are a first step at formalizing specifications for system design.

SUMMARY

Boehm has stated seven principles that have helped organize the techniques discussed in this paper [BOEH76].

1) *Manage using a sequential life cycle plan.* This means to follow the software development life cycle outlined earlier. It allows for feedback which updates previous stages as the consequences of previous decisions become unknown. It encourages milestones to measure progress.

2) *Perform continuous validation.* Certify each new refinement of a module. Use walkthroughs and code reading. Display the hierarchical structure of the system clearly in all documentation.

3) *Maintain disciplined product control.* All output of a project—design documents, source code, user documentation, and so forth—should be formally approved. Changes to documents and program libraries must be strictly monitored and audited. Code reading, project reporting forms, librarians, a development library, and a project notebook all contribute to this goal.

4) *Use enhanced top-down structured programming.* PL/I and PASCAL have good control and data structures. Preprocessors exist which augment FORTRAN for these structures. Description techniques such as stepwise refinement, nested data abstractions, and data flow networks should be used.

5) *Maintain clear accountability.* Use milestones to measure progress, and a project notebook to monitor each individual's efforts.

6) *Use better and fewer people.* The chief programmer team, in which each individual is skilled and accountable for his actions, and good results are rewarded, aids in this effort.

7) *Maintain commitment to improve process.* Settle only for the best; strive for improvement. Be open to new develop-

ments in software engineering, but do not sacrifice reliability for modifiability while pursuing them.

Progress has been made in understanding how large-scale software systems are built, yet more needs to be done. Management aids must be improved and project control techniques developed. The role of software management is coming more to resemble that of engineering management in other disciplines. We can no longer afford costly mistakes when systems are so large and we depend so much on them. Most importantly, we must be patient; we need to gain experience on which future theories can rely.

ACKNOWLEDGMENTS

The author is indebted to Peter Denning for his detailed review and to the referees for their valuable comments on this paper. This work was partially supported by grant number DCR 74-11520-AO1 from the National Science Foundation to the National Bureau of Standards.

REFERENCES

[AHO72] AHO, A.; AND ULLMAN, J. *Theory of parsing, translation, and compiling,* Prentice Hall, Inc., Englewood Cliffs, N J., 1972.

[ANSI66] *American Standard FORTRAN,* American Natl. Standards Inst., x3.9-1966, March, 1966.

[ANSI76] *American Standard PL/1,* American Natl. Standards Inst., x.53-1976, Aug., 1976.

[BAKE72] BAKER, F. T. "Chief programmer team management of production programming," *IBM Syst. J.* **11,** 1 (1972), 56-73.

[BASI78] BASILI, V.; AND ZELKOWITZ, M. "Analyzing medium scale software development," *Third Int. Conf. Software Engineering,* 1978.

[BASI75] BASILI, V.; AND TURNER, A. J. "Iterative enhancement: a practical technique for software development," *IEEE Trans. Softw. Eng.* **1,** 4 (Dec. 1975), 390-396.

[BOEH75] BOEHM, B.; MCCLEAN, R.; AND URFRIG, D. "Some experience with automated aids to the design of large scale reliable software," *Int. Conf. on Reliable Software,* 1975, ACM, New York, pp. 105-113.

[BOEH77] BOEHM, B. "Seven basic principles of software engineering," in *Infotech state of the art report on software engineering techniques,* 1977, Infotech International Ltd., Maidenhead, UK, 1976.

[BRIN77] BRINCH, HANSEN P. *Architectures of concurrent programs,* Prentice Hall, Inc., Englewood Cliffs, N. J., 1977.

[BROO75] BROOKS, F. P. *The mythical man month,* Addison-Wesley Publ Co., Reading, Mass., 1975.

[CAIN75] CAINE, S. H.; AND GORDON, E. K. "PDL—a tool for software design," in *Proc. 1975 AFIPS Natl. Computer Conf.,* Vol. 44, AFIPS Press, Montvale, N. J., pp. 271-276.

[CONW78] CONWAY, R. *A primer on disciplined programming,* Winthrop Publishers, Cambridge, Mass., 1978.

[CONW73] CONWAY, R.; AND WILCOX, T. "Design and implementation of a diagnostic compiler for PL/1," *Commun. ACM* **16,** 3 (March 1973), 169-179.

[COOL65] COOLEY, J. W.; AND TUKEY, J. W. "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* **19,** 90 (1965), 299-301.

[DAVI77] DAVIS, C. G.; AND VICK, C. R. "The software development system," *IEEE Trans. Softw. Eng.* **3,** 1 (Jan., 1977), 69-84.

[DENN76a] DENNING, P. J. "A hard look at structured programming," in *Infotech state of the art report on structured programming,* 1976, Infotech International Ltd., Maidenhead, UK, pp. 183-202.

[DENN76b] DENNING, P. J. "Fault tolerant operating systems," *Comput. Surv.* **8,** 4 (Dec. 1976), 359-389.

[DIJK68] DIJKSTRA, E. "GOTO statement considered harmful," *Commun. ACM* **11,** 3 (March 1968), 147-148.

[DIJK76] DIJKSTRA, E. *A discipline of programming,* Prentice Hall, Inc., Englewood Cliffs, N. J., 1976.

[DOLO76] DOLOTTA, T. A.; AND MASHEY, J. R. "An introduction to the programmer's workbench," in *Second Int. Conf Software Engineering,* 1976, pp. 164-168.

[ENR61] "Everything about the Narrows Bridge is big, bigger, or biggest," *Eng. News Record* 166, June 29, 1961, 24-28.

[ENR64] "Narrows Bridge opens to traffic," *Eng. News Record* 173, Nov. 19, 1964, 33.

[ENR77] "Alaskan pipe cost probe hits snag," *Eng. News Record* 198, April 7, 1977, 14.

[FIFE77] FIFE, D. *Computer software management: a primer for project management and quality control,* Natl. Bureau of Standards, Inst. Computer Sciences and Technology, Special Publications, April 1977.

[GALL65] GALLAGHER, P. F. *Project estimating by engineering methods,* Hayden Book Co., New York, 1965.

[GERH76] GERHART, S.; AND YELOWITZ, L. "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Softw. Eng.* **2,** 3 (Sept. 1976), 195-207.

[GOOD75] GOODENOUGH, J. B.; AND GERHART, S. "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.* **1,** 2 (June 1975), 156-173.

[HALS77] HALSTEAD, M. *Elements of software science,* Elsevier North Holland, Inc., New York, 1977.

[HAMI76] HAMILTON, M.; AND ZELDIN, S. "Higher order software—a methodology for defining software," *IEEE Trans. Softw. Eng.* **2,** 1 (March 1976), 9-32.

[HOAR69] HOARE, C. A. R. "An axiomatic basis for computer programming," *Commun.*

ACM **12,** 10 (Oct. 1969), 576–580, 583.

[HUAN75] HUANG, J. C. "An approach to program testing," *Comput. Surv.* **7,** 3 (Sept. 1975), 113–128.

[JEFF77] JEFFERY, S., AND LINDEN, T. "Software engineering is engineering," in *IEEE Computer Science and Engineering Curricula Workshop,* 1977, IEEE, New York, pp. 112–115.

[KING69] KING, J. C. "A program verifier," PhD Dissertation, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, Pa., 1969.

[KNUT71] KNUTH, D. "An empirical study of FORTRAN programs." *Softw. Pract Exper.* **1,** 2 (1971), 105–133.

[KNUT74] KNUTH, D. "Structured programming with statements," *Comput Surv.* **6,** 4 (Dec. 1974), 261–301.

[LISK75] LISKOV, B.; AND ZILLES, S. "Specification techniques for data abstractions," *IEEE Trans. Softw Eng.* **1,** 1 (1975), 9–19.

[LISK77] LISKOV, B.; SNYDER, A., ATKINSON, R.; AND SCHAFFERT, C. "Abstraction mechanisms in CLU," *Commun. ACM* **20,** 8 (Aug. 1977), 564–576.

[LOWR69] LOWRY, E. S.; AND MEDLOCK, C. W. "Object code optimization," *Commun. ACM* **12,** 1 (Jan. 1969), 13–22.

[MILL76] MILLS, H. D. "Software development," *IEEE Trans Softw. Eng.* **2,** 4 (1976), 265–273

[PARN72] PARNAS, D. L. "On the criteria for decomposing systems into modules," *Commun. ACM* **15,** 12 (Dec. 1972), 1053–1058.

[PARN75] PARNAS, D. L. "The influence of software structure on reliability," in *Int. Conf. Reliable Software,* 1975, pp. 358–362, (*ACM SIGPLAN Notices* **10,** 6 June 1975).

[POPE77] POPEK, G. J.; HORNING, J. J.; LAMPSON, B. W.; MITCHELL, J. G.; AND LONDON, R. L. "Notes on the design of EUCLID," in *Proc ACM Conf Language Design for Reliable Software,* ACM, New York, 1977, pp. 11–18.

[PUTN77] PUTNAM, L.; AND WOLVERTON, R. *Quantitative management. software cost estimating,* (tutorial), IEEE Computer Society, Nov. 1977, IEEE, New York..

[RAMA75] RAMAMOORTHY, C. V., AND HO, S. F. "Testing large software with automated software evaluation systems," *IEEE Trans. Softw Eng.* **1,** 1 (1975), 46–58.

[REIF76] REIFER, D. J. "The structured FORTRAN dilemma," *SIGPLAN Notices* **11,** 2 (1976), 30–32.

[RITC74] RITCHIE, D. M.; AND THOMPSON, K. "The UNIX time-sharing system," *Commun ACM* **17,** 7 (July 1974), 365–375

[TEIC77] TEICHROEW, D.; AND HERSHEY, E. A. "PSL/PSA. a computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans Softw Eng.* **3,** 1 (1977), 41–48.

[WALS77] WALSTON, C. E., AND FELIX, C. P. "A method of programming measurements and estimation," *IBM Syst. J.* **16,** 1 (1977), 54–73.

[WEID77] WEIDE, B. "A survey of analysis techniques for discrete algorithms," *Comput. Surv.* **9,** 4 (Dec. 1977), 291–313

[WEIN71] WEINBERG, G. M. *The psychology of computer programming,* Van Nostrand Reinhold, New York, 1971.

[WIRT71] WIRTH, N. "Program development by stepwise refinement," *Commun. ACM* **14,** 4 (April 1971), 221–227.

[WIRT74] WIRTH, N. "On the composition of well-structured programs," *Comput. Surv* **6,** 4 (Dec 1974), 247–259

[WOLV74] WOLVERTON, R. W. "The cost of developing large scale software," *IEEE Trans. Comput.* **23,** 6 (1974), 615–636.

[WULF76] WULF, W.; LONDON, R., SHAW, M. "An introduction to the construction and verification of ALPHARD programs," *IEEE Trans Softw. Eng.* **2,** 4 (1976), 253–264.

[YOUN67] YOUNGER, D. "Recognition and parsing of context-free languages in time n**3," *Inf Control* **10,** 2 (1967), 189–208.

[ZELK75] ZELKOWITZ, M. V. "Third generation compiler design," in *ACM Natl. Comput Conf.,* 1975, ACM, New York, pp. 253–258.

RECEIVED MARCH 14, 1977; FINAL REVISION ACCEPTED MARCH 7, 1978