CMU-CS-80-116

The Impact of Abstraction Concerns on Modern Programming Languages

Mary Shaw

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa.

April 1980

Abstract

The major issues of modern software are its size and complexity, and its major problems involve finding effective techniques and tools for organization and maintenance. This paper traces the important ideas of modern programming languages to their roots in the problems and languages of the past decade and shows how these modern languages respond to contemporary problems in software development. Modern programming's key concept for controlling complexity is *abstraction* -- that is, selective emphasis on detail; new developments in programming languages provide ways to support and exploit abstraction techniques.

To appear in *Proceedings of the IEEE* in September 1980.

This research was sponsored by the National Science Foundation under Grant MCS77-03883 and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, or the US Government.

Table of Contents

List of Figures

1. Issues of Modern Software

The major issues of modern software development stem from the costs of software development, use, and maintenance -- which are too high -- and the quality of the resulting systems -- which is too low. These problems are particularly severe for the large complex programs with long useful lifetimes that characterize modern software. Such programs typically involve many programmers, not only during their development but also for maintenance and enhancement after they are initially released. As a result, the cost and quality of software are influenced by both management and software engineering considerations [5] [23].

This paper examines one of the themes that run through the history of attempts to solve the problems of high cost and low quality: the effect of abstraction techniques and their associated specification and verification issues on the evolution of modern programming languages and methods. This theme places a strong emphasis on engineering concerns, including design, specification, correctness, and reliability.

The paper begins with a review of the ideas about program development and analysis that heavily influenced the development of current techniques (Section 2). Many of these ideas are of current interest as well as of historical importance. This review provides a setting for a survey of the ideas from current research projects that are influencing modern language design and software methodology (Section 3). Section 4 illustrates the changes in program organization that have been stimulated by this work by developing an example in two different languages intended for production use, Ada and Pascal. Although Sections 2 and 3 present a certain amount of technical detail, Section 4 illustrates the concepts with an example that should be accessible to all readers. An assessment of the current status and the potential of current abstraction techniques (Section 5) concludes the paper.

2. Historical Review of Abstraction Techniques

Controlling software development and maintenance has always involved managing the intellectual complexity of programs and systems of programs. Not only must the systems be created, they must be tested, maintained, and extended. As a result, many different people must understand and modify them at various times during their lifetimes. This section identifies one set of ideas about managing program complexity and shows how those ideas have shaped programming languages and methodologies over the past ten to fifteen years.

A dominant theme in the evolution of methodologies and languages is the development of tools for dealing with abstractions. An *abstraction* is a simplified description, or *specification*, of a system that emphasizes some of the system's details or properties while suppressing others. A *good* abstraction

is one in which information that is significant to the reader (i.e., the user) is emphasized while details that are immaterial or diversionary, at least for the moment, are suppressed.

What we call "abstraction" in programming systems corresponds closely to what is called "analytic modelling" in many other fields. It shares many of the same problems: deciding which characteristics of the system are important, what variability (i.e., parameters) should be included, which descriptive formalism to use, how the model can be validated, and so on. As in many other fields, we often define hierarchies of models in which lower-level models provide more detailed explanations for the phenomena that appear in higher-level models. Our models also share the property that the description is sufficiently different from the underlying system to require explicit validation. We refer to the abstract description of a model as its *specification* and to the next lower-level model in the hierarchy as its *implementation*. The validation that the specification is consistent with the implementation is called *verification*. The abstractions we use for software tend to emphasize functional properties of the software, emphasizing *what* results are to be obtained and suppressing details about *how* this is to be achieved.

Many important techniques for program and language organization have been based on the principle of abstraction. These techniques have evolved in step not only with our understanding of programming issues, but also with our ability to use the abstractions as *formal specifications* of the systems they describe. In the 1960's, for example, the important developments in methodology and languages were centered around functions and procedures, which summarize a program segment in terms of a name and a parameter list. At that time, we only knew how to perform syntactic validity checks, and specification techniques reflected this: "specification" meant little more than "procedure header" until late in the decade. By the late 1970's, developments were centered on the design of data structures, specification techniques drew on quite sophisticated techniques of mathematical logic, and programming language semantics were well enough understood to permit formal verification that these programs and specifications were consistent.

Programming languages and methodologies often develop in response to new ideas about how to cope with complexity in programs and systems of programs. As languages evolve to meet these ideas, we reshape our perceptions of the problems and solutions in response to the new experiences. Our sharpened perceptions in turn generate new ideas which feed the evolutionary cycle. This paper explores the routes by which these cyclic advances in methodology and specification have led to current concepts and principles of programming languages.

2.1. Early Abstraction Techniques

Prior to the late 1960's, the set of programming topics regarded as important was dominated by the syntax of programming languages, translation techniques, and solutions to specific implementation

problems. Thus we saw many papers on solutions to specific problems such as parsing, storage allocation, and data representation. Procedures were well-understood, and libraries of procedures were set up. These libraries met with mixed success, often because the documentation (informal specification) was inadequate or because the parameterization of the procedures did not support the cases of interest. Basic data structures such as stacks and linked lists were just beginning to be understood, but they were sufficiently unfamiliar that it was difficult to separate the concepts from the particular implementations. Perhaps it was too early in the history of the field for generalization and synthesis to take place, but in any event abstraction played only a minor role.

Abstraction was first treated consciously as a program organization technique in the late 1960's. Earlier languages supported built-in data types including at least integers, real numbers, and arrays, and sometimes booleans, high-precision reals, etc. Data structures were first treated systematically in 1968 (the first edition of [43]), and the notion that a programmer might define data types tailored to a particular problem first appeared in 1967 (e.g., [67]). Although discussions of programming techniques date back to the beginning of the field, the notion that programming is an activity that should be studied and subjected to some sort of discipline dates to the NATO Software Engineering conferences of 1968 [53] and 1969 [7].

2.2. Extensible Languages

The late 1960's also saw efforts to abstract from the built-in notations of programming languages in such a way that any programmer could add new notation and new data types to a base language. The objectives of the extensible language work included allowing individual programmers to extend the syntax of the programming language, to define new data structures, to add new operators (including infix operators as well as ordinary functions) for both old and new data structures, and to add new control structures to the base language. This work on extensibility [60] died out, in part because it underestimated the difficulty of defining interesting extensions. The problems included difficulty with keeping independent extensions compatible when all of them modify the syntax of the base language, with organizing definitions so that related information was grouped in common locations, and with finding techniques for describing an extension accurately (other than by exhibiting the code for the extension). However, it left a legacy in its influence on the abstract data types and generic definitions of the 1970's.

2.3. Structured Programming

By the early 1970's, a methodology emerged for constructing programs by progressing from a statement of the objective through successively more precise intermediate stages to final code [71] [17]. Called "stepwise refinement" or "top-down programming", this methodology involves approaching a problem by writing a program that is free to assume the existence of any data

structures and operations that can be directly applied to the problem at hand, even if those structures and operations are quite sophisticated and difficult to implement. Thus the initial program is presumably small, clear, directly problem-related, and "obviously" correct. Although the assumed structures and operations may be specified only informally, the programmer's intuitions about them should make it possible to concentrate on the overall organization of the program and defer concerns about the implementations of the assumed definitions. When each of the latter definitions is addressed, the same technique is applied again, and the implementations of the high-level operations are substituted for the corresponding invocations. The result is a new, more detailed program that is convincingly like the previous one, but depends on fewer or simpler definitions (and hence is closer to being compilable). Successive steps of the program development add details of the sort more relevant to the programming language than to the problem domain until the program is completely expressed using the operations and data types of the base language, for which a compiler is available.

This separation of concerns between the structures that are used to solve a problem and the way those structures are implemented provides a methodology for decomposing complex problems into smaller, fairly independent segments. The key to the success of the methodology is the degree of abstraction imposed by selecting high-level data structures and operations. The chief limitation of the methodology, which was not appreciated until the methodology had been in use for some time, is that the final program does not preserve the series of abstractions through which it was created, and so the task of modifying the program after it is completed is not necessarily simpler than it would be for a program developed in any other way. Another limitation of the methodology is that informal descriptions of operations do not conveyprecise information. Misunderstandings about exactly what an operation is supposed to do can complicate the program development process, and informal descriptions of procedures may not be adequate to assure true independence of modules. The development of techniques for formal program specification helps to alleviate this set of problems.

At about the same time as this methodology was emerging, we also began to be concerned about how people understand programs and how programs can be organized to make them easier to understand, and hence to modify. We realized that it is of primary importance to be able to determine what assumptions about the program state are being made at any point in the program. Further, arbitrary transfers of control that span large amounts of program text interfere with this goal. The control flow patterns that lend themselves to understandable programs are the ones that have a single entry point (at the beginning of the text) and, at least conceptually, a single exit point (at the end of the text). Examples of statements that satisfy this rule are the **if...then...else** and the **for** and **while** loops. The chief violator of the rule is the **go to** statement.

The first discussion of this question appeared in 1968 [16], and we converged on a common set of

"ideal" control constructs a few years later [17] [35]. Although true consensus on this set of constructs has still not been achieved, the question is no longer regarded as an issue.

2.4. Program Verification

In parallel with the development of "ideal" control constructs -- in fact, as part of their motivation -- computer scientists became interested in finding ways to make precise, mathematically manipulatable statements about what a program computes. The ability to make such statements is essential to the development of techniques for reasoning about programs, particularly for techniques that rely on abstract specifications of effects. New techniques were required because procedure headers, even accompanied by prose commentary, provide inadequate information for reasoning precisely about programs, and imprecise statements lead to ambiguities about responsibilities and inadequate separation of modules.

The notion that it is possible to make formal statements about values of variables (a set of values for the variables of a program is called the *program state*) and to reason rigorously about the effect of executing a statement on the program's state first appeared in the late 1960's [19] [32]. The formal statements are expressed as formulas in the predicate calculus, such as

$$y>x \wedge (x>0 \supset z = x^2).$$

A programming language is described by a set of rules that define the effect each statement has on the logical formula that describes the program state. The rules for the language are applied to the assertions in the program in order to obtain theorems whose proofs assure that the program matches the specification.[1] By the early 1970's the basic concepts of verifying assertions about simple programs and describing a language in such a way that this is possible were under control [35] [48]. When applied by hand, verification techniques tend to be error-prone, and formal specifications, like informal ones, are susceptible to errors of omission [20]. In response to this problem, systems for performing the verification steps automatically have been developed [21]. Verification requires converting a program annotated with logical assertions to logical theorems with the property that the program is correct if and only if the theorems are true. This conversion process, called *verification condition generation*, is well-understood, but considerable work remains to be done on the problem of proving those theorems.

When the emphasis in programming methodology shifted to using data structures as a basis for program organization, corresponding problems arose for specification and verification techniques. The initial efforts addressed the question of what information is useful in a specification [55]. Subsequent attention concentrated on making those specifications more formal and dealing with the

[1] A survey of these ideas appears in [47]; introductions to the methods appear in Chapter 3 of [49] and Chapter 5 of [75].

verification problems [33]. From this basis, work on verification for abstract data types proceeded as described in Section 3.

2.5. Abstract Data Types

In the 1970's we recognized the importance of organizing programs into modules in such a way that knowledge about implementation details was localized as much as possible. This led to language support for data types [34], for specifications that are organized using the same structure as data [28] [44] [74], and for generic definitions [61]. The language facilities are based on the **class** construct of Simula [8] [9], ideas about strategies for defining modules [54] [56], and concerns over the impact of locality on program organization [73]. The corresponding specification techniques include strong typing and verification of assertions about functional correctness.

Over the past five years, most research activity in abstraction techniques has been focussed on the language and specification issues raised by these considerations; much of the work is identified with the concept of *abstract data types*. Like structured programming, the methodology of abstract data types emphasizes locality of related collections of information. In this case, attention is focussed on data rather than on control, and the strategy is to form modules consisting of a data structure and its associated operations. The objective is to treat these modules in the same way as ordinary types such as integers and reals are treated; this requires support for declarations, infix operators, specification of routine parameters, and so on. The result, called an *abstract data type*, effectively extends the set of types available to a program -- it explains the properties of a new group of variables by specifying the values one of these variables may have, and it explains the operations that will be permitted on the variables of the new type by giving the effects these operations have on the values of the variables.

In a data type abstraction, we specify the functional properties of a data structure and its operations, then we implement them in terms of existing language constructs (and other data types) and show that the specification is accurate. When we subsequently use the abstraction, we deal with the new type solely in terms of its specification. (This technique is discussed in detail in section 3.) This philosophy was developed in several recent language research and development projects, including Ada [37], Alphard [74], CLU [46], Concurrent Pascal [4], Euclid [44], Gypsy [1], Mesa [22], and Modula [72].

The specification techniques used for abstract data types evolved from the predicates used in simple sequential programs. Additional expressive power was incorporated to deal with the way information is packaged into modules and with the problem of abstracting from an implementation to a data type [29]. One class of specification techniques draws on the similarity between a data type and the mathematical structure called an algebra [28] [45]. Another class of techniques explicitly

models a newly-defined type by defining its properties in terms of the properties of common, well-understood types [74].

In conjunction with the work on abstract data types and formal specifications, the generic definitions that originated in extensible languages have been developed to a level of expressiveness and precision far beyond the anticipation of their originators. These definitions, discussed in detail in Section 3.3, are parameterized not only in terms of variables that can be manipulated during program execution, but also in terms of data types. They can now describe restrictions on which types are acceptable parameters in considerable detail, as in [2].

2.6. Interactions Between Abstraction and Specification Techniques

As this review shows, programming languages and methodologies evolve in response to the needs that are perceived by software designers and implementors. However, these perceived needs themselves evolve in response to experience gained with past solutions. The original abstraction techniques of structured programming were procedures or macros;[2] these have evolved to abstract types and generic definitions. Methodologies for program development emerge when we find common useful patterns and try to use them as models; languages evolve to support these methodologies when the models become so common and stable that they are regarded as standard. A more extensive review of the development of software abstractions appears in [26]. As abstraction techniques have become capable of addressing a wider range of program organizations, formal specification techniques have become more precise and have played a more crucial role in the programming process.

For an abstraction to be used effectively, its specification must express all the information needed by the programmer who uses it. Initial attempts at specification used the notation of the programming language to express things that could be checked by the compiler: the name of a routine and the number and types of its parameters. Other facts, such as the description of what the routine computed and under what conditions it should be used, were expressed informally [76]. We have now progressed to the point that we can write precise descriptions of many important relations among routines, including their assumptions about the values of their inputs and the effects they have on the program state. However, many other properties of abstractions are still specified only informally. These include time and space consumption, interactions with special-purpose devices, very complex aggregate behavior, reliability in the face of hardware malfunctions, and many aspects of concurrent processing. It is reasonable to expect future developments in specification techniques and programming languages to respond to those issues.

---

[2] Although procedures were originally viewed as devices to save code space, they soon came to be regarded, like macros, as abstraction tools.

The history of programming languages shows a balance between language ideas and formal techniques; in each methodology, the properties we specify are matched to our current ability to validate (verify) the consistency of a specification and its implementation. Thus, since we can rely on formal specifications only to the extent that we are certain that they match their implementations, the development of abstraction techniques, specification techniques, and methods of verifying the consistency of a specification and an implementation must surely proceed hand in hand. In the future, we should expect to see more diversity in the programs that are used as a basis for modularization; we should also expect to see specifications that are concerned with aspects of programs other than the purely functional properties we now consider.

3. Abstraction Facilities in Modern Programming Languages

With the historical background of Section 2, we now turn to the abstraction methodologies and specification techniques that are currently under development in the programming language research community. Some of the ideas are well enough worked out to be ready for transfer to practical languages, but others are still under development.

Although the ideas behind modern abstraction techniques can be explored independently of programming languages, the instantiation of these ideas in actual languages is also important. Programming languages are our primary notational vehicle for expressing a class of very complex ideas; the concepts we must deal with include not only the functional relations of mathematics, but also constructs that deal with relations over time, such as sequentiality and synchronization. Language designs influence the ways we think about algorithms by making some program structures easier to describe than others. In addition, programming languages are used for communication among people as well as for controlling machines. This role is particularly important in long-lived programs, because a program is in many ways the most practical medium for expressing the structure imposed by the designer -- and for maintaining the accuracy of this documentation over time. Thus, even though most programming languages technically have the same expressive power, differences among languages can significantly affect their practical utility.

3.1. The New Ideas

Current activity in programming languages is driven by three sets of global concerns: simplicity of design, the potential for applying precise analytic techniques to formal specifications, and the need to control costs over the entire lifetime of a long-lived program.

Simplicity has emerged as a major criterion for evaluating programming language designs. We see a certain tension between the need for "just the right construct" for a task and the need for a language small enough to understand thoroughly. This is an example of a tradeoff between

specialization and generality: if highly specialized constructs are provided, individual programs will be smaller, but at the expense of complexity (and feature-by-feature interactions) in the system as a whole. The current trend is to provide a relatively small base language that provides facilities for defining special facilities in a regular way [65]. An emphasis on simplicity underlies a number of design criteria that are now commonly used. When programs are organized to localize information, for example, assumptions shared among program parts and module interfaces can be significantly simplified. The introduction of support for abstract data types in programming languages allows programmers to design special-purpose structures and deal with them in a simple way; it does so by providing a definition facility that allows the extensions to be made in a regular, predictable fashion. The regularity introduced by using these facilities can substantially reduce maintenance problems by making it easier for a programmer who is unfamiliar with the code to understand the assumptions about the program state that are made at a given point in the program -- thereby increasing the odds that he or she can make a change without introducing new errors.

Our understanding of the principles underlying programming languages has improved to the point that *formal and quantitative techniques* are both feasible and useful. Current methods for specifying properties of abstract data types and for verifying that those specifications are consistent with the implementation are discussed in Section 3.2. Critical studies of testing methods are being performed [36], and interest in quantitative methods for evaluating programs is increasing [58]. It is interesting to note that there seems to be a strong correlation between the ease with which proof rules for language constructs can be written and the ease with which programmers can use those constructs correctly and understand programs that use them.

The 1970's mark the beginning of a real appreciation that the cost of software includes the *costs over the lifetime of the program*, not just the costs of initial development or of execution. For large, long-lived programs, the cost of enhancement and maintenance usually dominate design, development, and execution costs, often by large factors. Two classes of issues arise [15]. First, in order to modify a program successfully, a programmer must be able to determine what other portions of the program depend on the section about to be modified. The problem of making this determination is simplified if the information is localized and if the design structure is retained in the structure of the program. Off-line design notes or other documents are not an adequate substitute except in the unlikely case that they are meticulously (and correctly) updated. Second, large programs rarely exist in only one version. The major issues concerning the control of large-scale program development are problems of management, not of programming. Nevertheless, language-related tools can significantly ease the problems. Tools are becoming available for managing the interactions among many versions of a program.

3.2. Language Support for Abstract Data Types

Over the past five years, the major thrust of research activity in programming languages and methodology has been to explore the issues related to abstract data types. The current state has emerged directly from the historical roots described in Section 2.5. The methodological concerns included the need for information hiding [54] [56] and locality of data access [73], a systematic view of data structures [34], a program organization strategy exemplified by the Simula **class** construct [8] [9], and the notion of generic definition [61]. The formal roots included a proposal for abstracting properties from an implementation [33] and a debate on the philosophy of types, which finally led to the view that types share the formal characteristics of abstract algebras [27] [28] [45] [51].

Whereas structured programming involves progressive development of a program by adding detail to its control structure, programming with abstract data types involves partitioning the program *in advance* into modules that correspond to the major data structures of the final system. The two methodologies are complementary, because the techniques of structured programming may be used within type definition modules, and conversely. An example of the interaction of the two design styles appears in [6].

In most languages that provide the facility, the definition of an abstract data type consists of a program unit that includes the following information:

- *visible outside the type definition*: the name of the type and the names and routine headers of all operations (procedures and functions) that are permitted to use the representation of the type; some languages also include formal specifications of the values that variables of this type may assume and of the properties of the operations.

- *not visible outside the type definition*: the representation of the type in terms of built-in data types or other defined types, the bodies of the visible routines, and hidden routines that may be called only from within the module.

An example of a module that defines an abstract data type appears in Figure 4-5.

The general question of abstract data types has been addressed in a number of research projects. These include Alphard [74], CLU [46], Gypsy [1], Russell [12], Concurrent Pascal [4], and Modula [72]. Although they differ in detail, they share the goal of providing language support adequate to the task of abstracting from data structures to abstract data types and allowing those abstract definitions to hold the same status as built-in data types. Detailed descriptions of the differences among these projects are best obtained by studying them in more detail than is appropriate here. As with many research projects, the impact they have is likely to take the form of influence on other languages rather than complete adoption. Indeed, the influence of several of the research projects on Ada [37] and Euclid [44] is apparent.

Programming with abstract data types requires support from the programming language, not

simply managerial exhortations about program organization. Suitable language support requires solutions to a number of technical issues involving both design and implementation. These include:

- *Naming*: Scope rules are required to ensure the appropriate visibility of names. In addition, protection mechanisms [41] [52] should be considered in order to guarantee that hidden information remains private. Further, programmers must be prevented from naming the same data in more than one way ("aliasing") if current verification technology is to be relied upon.

- *Type checking*: It is necessary to check actual parameters to routines, preferably during compilation, to be sure they will be acceptable to the routines. The problem is more complex than the type checking problem for conventional languages because new types may be added during the compilation process and the parameterization of types requires subtle decisions in the definition of a useful type checking rule.

- *Specification notation*: The formal specifications of an abstract data type should convey all information needed by the programmer. This is not yet possible, but current progress is described below. As for any specification formalism, it is also necessary to develop a method for verifying that a specification is consistent with its implementation.

- *Distributed properties*: In addition to providing operations that are called as routines or infix operators, abstract data types must often supply definitions to support type-specific interpretation of various constructs of the programming language. These constructs include storage allocation, loops that operate on the elements of a data structure without knowledge of the representation, and synchronization. Some of these have been explored, but many open questions remain [46] [62] [65].

- *Separate compilation*: Abstract data types introduce two new problems to the process of separate compilation. First, type checking should be done across compilation units as well as within units. Second, generic definitions offer significant potential for optimization (or for inefficient implementation).

Specification techniques for abstract data types are the topic of a number of current research projects. Techniques that have been proposed include informal but precise and stylized English [31], models that relate the new type to previously defined types [74], and algebraic axioms that specify new types independently of other types [27]. Many problems remain. The emphasis to date has been on the specification of properties of the code; the correspondence of these specification to informally understood requirements is also important [11]. Further, the work to date has concentrated almost exclusively on the functional properties of the definition without attending, for example, to the performance or reliability.

Not all the language developments include formal specifications as part of the code. For example, Alphard includes language constructs that associate a specification with the implementation of a module; Ada and Mesa expect interface definitions that contain at least enough information to support separate compilation. All the work, however, is based on the premise that the specification must include all information that should be available to a user of the abstract data type. When it has

been verified that the implementation performs in accordance with its public specification [33], the abstract specification may safely be used as the definitive source of information about how higher-level programs may correctly use the module. In one sense we build up "bigger" definitions out of "smaller" ones; but because a specification alone suffices for understanding, the new definition is in another sense no bigger than the pre-existing components. It is this regimentation of detail that gives the technique its power.

3.3. Generic Definitions

A particularly rich kind of abstract data type definition allows one abstraction to take another abstraction (e.g., a data type) as a parameter. These *generic* definitions provide a dimension of modelling flexibility that conventionally-parameterized definitions lack.

For example, consider the problem of defining data types for an application that uses three kinds of unordered sets: sets of integers, sets of reals, and sets of a user-defined type for points in 3-dimensional space. One alternative would be to write a separate definition for each of these three types. However, that would involve a great deal of duplicated text, since both the specifications and the code will be very similar for all the definitions. In fact, the programs would probably differ only where specific references to the types of set elements are made, and the machine code would probably differ only where operations on set elements (such as the assignment used to store a new value into the data structure) are performed. The obvious drawbacks of this situation include duplicated code, redundant programming effort, and complicated maintenance (since bugs must be fixed and improvements must be made in all versions).

Another alternative would be to separate the properties of unordered sets from the properties of their elements. This is possible because the definition of the sets relies on very few specific properties of the elements -- it probably assumes only that ordinary assignment and equality operations for the element type are defined. Under that assumption, it is possible to write a single definition, say

```
type UnOrderedSet(T: type) is ...
```

that can be used to declare sets with several different types of elements, as in

```
var
        Counters: UnOrderedSet(integer);
        Timers: UnOrderedSet(integer);
        Sizes: UnOrderedSet(real);
        Places: UnOrderedSet(PointIn3Space);
```

using a syntax appropriate to the language that supports the generic definition facility. The definition of UnOrderedSet would provide operations such as Insert, TestMembership, and so on; the declarations of the variables would instantiate versions of these operations for all relevant element

types, and the compiler would determine which of the operations to use at any particular time by inspecting the parameters to the routines.

The flexibility provided by generic definitions is demonstrated by the algorithmic transformation of [2], which automatically converts any solution of one class of problems to a solution of the corresponding problem in a somewhat larger class. This generic definition is notable for the detail and precision with which the assumptions about the generic parameter can be specified.

4. Practical Realizations

A number of programming languages provide some or all of the facilities required to support abstract data types. In addition to implementations of research projects, several language efforts have been directed primarily at providing practical implementations. These include Ada [37], Mesa [22], Pascal [40], and Simula [8]. Of these, Pascal currently has the largest user community, and the objective of the Ada development has been to make available a language to support most of the modern ideas about programming. Because of the major roles they play in the programming language community, Pascal and Ada will be discussed in some detail.

4.1. A Small Example Program

In order to illustrate the effects that modern languages have on program organization and programming style, we will carry a small example through the discussion. This section presents a Fortran program for the example; Pascal and Ada versions are developed in Section 4.2 and 4.3.

The purpose of the program is to produce the data needed to print an internal telephone list for a division of a small company. A data base containing information about all employees, including their names, divisions, telephone numbers, and salaries is assumed to be available. The program must produce a data structure containing a sorted list of the employees in a selected division and their telephone extensions.

Suitable declarations of the employee data base and the divisional telephone list for the Fortran implementation are given in Figure 4-1. A program fragment for constructing the telephone list is given in Figure 4-2.

The employee data base is represented as a set of vectors, one for each unit of information about the employee. The vectors are used "in parallel" as a single data structure -- that is, part of the information about the i^{th} employee is stored in the i^{th} element of each vector. Similarly, the telephone list is constructed in two arrays, `DivNam` for names and `DivFon` for telephone numbers.

The telephone list is constructed in two stages. First, the data base is scanned for employees

```
c       Vectors that contain Employee information
c       Name is in EmpNam (24 chars), Phone is in EmpFon (integer)
c       Salary in in EmpSal (real), Division is in EmpDiv (4 chars)
        integer EmpFon(1000), EmpDiv(1000)
        real EmpSal(1000)
        double precision EmpNam(3,1000)

c       Vectors that contain Phone list information
c       Name is in DivNam (24 chars), Phone is in DivFon (integer)
        integer DivFon(1000)
        double precision DivNam(3,1000)

c       declarations of scalars used in program
        integer StafSz, DivSz, i, j
        integer WhichD
        double precision q
```

Figure 4-1: Declarations for Fortran Version of Telephone List Program

whose division (`EmpDiv(i)`) matches the division desired (`WhichD`). When a match is found, the name and phone number of the employee are added to the telephone list. Second, the telephone list is sorted using an insertion sort.[3]

There are several important things to notice about this program. First, the data about employees is stored in four arrays, and the relation among these arrays is shown only by the similar naming and the comment with their declarations. Second, the character string for each employee's name must be handled in eight-character segments, and there is no clear indication in either the declarations or the code that character strings are involved.[4] The six-line test that determines whether `DivNam(*,i) < DivNam(*,j)` could be reduced to three tests if it were changed to a test for less-than-or-equal, but this would make the sort unstable. Third, all the data about employees, including salaries, is easily accessible and modifiable; this is undesirable from an administrative standpoint.

4.2. Pascal

Pascal [40] is a simple algebraic language that was designed with three primary objectives. It was to support modern programming development methodology; it was to be a simple enough language to teach to students; and it was to be easy to implement reliably, even on small computers. It has, in

---

[3] This selection is not an endorsement of insertion sorting in general. However, most readers will recognize the algorithm, and the topic of this paper is the evolution of programming languages, not sorting techniques.

[4] Indeed, the implementations of floating point in some versions of Fortran interfere with this type violation. Character strings are dealt with more appropriately in the Fortran77 standard.

```
c       Get data for division WhichD only

        DivSz = 0
        do 200 i = 1,StafSz
            if (EmpDiv(i) .ne. WhichD) go to 200
            DivSz = DivSz + 1
            DivNam(1,DivSz) = EmpNam(1,i)
            DivNam(2,DivSz) = EmpNam(2,i)
            DivNam(3,DivSz) = EmpNam(3,i)
            DivFon(DivSz) = EmpFon(i)
200         continue

c       Sort telephone list

        if (DivSz .eq. 0) go to 210
        do 220 i = 1,DivSz
            do 230 j = i+1,DivSz
                if (DivNam(1,i) .gt. DivNam(1,j)) go to 240
                if (DivNam(1,i) .lt. DivNam(1,j)) go to 230
                if (DivNam(2,i) .gt. DivNam(2,j)) go to 240
                if (DivNam(2,i) .lt. DivNam(2,j)) go to 230
                if (DivNam(3,i) .gt. DivNam(3,j)) go to 240
                go to 230
240             do 250 k = 1,3
                    q = DivNam(k,i)
                    DivNam(k,i) = Divnam(k,j)
250                 DivNam(k,j) = q
                k = DivFon(i)
                DivFon(i) = DivFon(j)
                DivFon(j) = k
230             continue
220         continue
210     continue
```

Figure 4-2: Code for Fortran Version of Telephone List Program

general, succeeded in all three respects.

Pascal provides a number of facilities for supporting structured programming. It provides the standard control constructs of structured programming, and a formal definition [35] facilitates verification of Pascal programs. It supports a set of data organization constructs that are suitable for defining abstractions. These include the ability to define a list of arbitrary constants as an *enumerated type*, the ability to define heterogeneous **records** with individually named fields, data types that can be dynamically allocated and referred to by pointers, and the ability to name a data structure as a **type** (though not to bundle up the data structure with a set of operations).

The language has become quite widely used. In addition to serving as a teaching language for

undergraduates, it is used as an implementation language for micro-computers [3] and it has been extended to deal with parallel programming [4]. An international standardization effort is currently under way [39].

Pascal is not without its disadvantages. It provides limited support for large programs, lacking separate compilation facilities and block structure other than nested procedures. Type checking does not provide quite as much control over parameter passing as one might wish, and there is no support for the encapsulation of related definitions in such a way that they can be isolated from the remainder of the program. Many of the disadvantages are addressed in extensions, derivative languages, and the standardization effort.

```
type
        String = array [1..24] of char;
        ShortString = array [1..8] of char;
        EmpRec = record
            Name:String;
            Phone:integer;
            Salary:real;
            Division:ShortString;
            end;
        PhoneRec = record Name:String; Phone:integer; end;

var
        Staff: array [1..1000] of EmpRec;
        Phones: array [1..1000] of PhoneRec;
        StaffSize, DivSize,i,j: integer;
        WhichDiv: char;
        q: PhoneRec;
```

Figure 4-3: Declarations for Pascal Version of Telephone List Program

We can illustrate some of Pascal's characteristics by returning to the program for creating telephone lists. Suitable data structures, including both type definitions and data declarations, are shown in Figure 4-3. A program fragment for constructing the telephone list is given in Figure 4-4.

The declarations open with definitions of four types which are not predefined in Pascal. Two (`String` and `ShortString`) are generally useful, and the other two (`EmpRec` and `PhoneRec`) were designed for this particular problem.

The definition of `String` and `ShortString` as types permits named variables to be treated as single units: operations are performed on an entire string variable, not on individual groups of characters. This abstraction simplifies the program, but more importantly, it allows the programmer to concentrate on the algorithm that uses the strings as names, rather than on keeping track of the

individual fragments of a name. The difference between the complexity of the code in Figures 4-2 and 4-4 may not seem large, but when it is compounded over many individual composite structures with different representations, the difference can be large indeed. If Pascal allowed programmer-defined types to accept parameters, a single definition of strings that took the string length as a parameter could replace `String` and `ShortString`; Ada does allow this, and the change is made in the Ada program of Section 4.3.

```
{ Get data for division WhichDiv only }

DivSize := 0;
for i := 1 to StaffSize do
    if Staff[i].Division = WhichDiv then
        begin
        DivSize := DivSize + 1;
        Phones[DivSize].Name := Staff[i].Name;
        Phones[DivSize].Phone := Staff[i].Phone;
        end;

{ Sort telephone list }

for i := 1 to DivSize do
    for j := i+1 to DivSize do
        if Phones[i].Name > Phones[j].Name then
            begin
            q := Phones[i];
            Phones[i] := Phones[j];
            Phones[j] := q;
            end;
```

Figure 4-4: Code for Pascal Version of Telephone List Program

The type definitions for `EmpRec` and `PhoneRec` abstract from specific data items to the notions "record of information about an employee" and "record of information for a telephone list". Both the employee data base and the telephone list can thus be represented as vectors whose elements are records of the appropriate types.

The declarations of `Staff` and `Phones` have the effect of indicating that all the components are related to the same information structure. In addition, the definition is organized as a collection of records, one for each employee -- so the primary organization of the data structure is by employee. On the other hand, the data organization of the Fortran program was dominated by the arrays that correspond to the fields, and the employees were secondary.

Just as in the Fortran program, the telephone list is constructed in two stages (Figure 4-4). Note that Pascal's ability to operate on strings and records as single units has substantially simplified the

manipulation of names and the interchange step of the sort. Another notable difference between the two programs is in the use of conditional statements. In the Pascal program, the use of **if ... then** statements emphasizes the conditions that will cause the bodies of the *if* statements to be executed. The Fortran **if** statements with **go to**'s, however, describe conditions in which code is *not* to be executed, leaving the reader of the program to compute the conditions that actually correspond to the actions.

It is also worth mentioning that the Pascal program will not execute the body of the sort loop at all if no employees work in division `WhichDiv` (that is, if `DivSize` is 0). The body of the corresponding Fortran loop would be executed once in that situation if the loop had not been protected by an explicit test for an empty list. While it would do no harm to execute this particular loop once on an empty list, in general it is necessary to guard Fortran loops against the possibility that the upper bound is less than the lower bound.

4.3. Ada

The Ada language is currently being developed under the auspices of the Department of Defense in an attempt to reduce the software costs of embedded computer systems. The project includes components for both a language and a programming support environment. The specific objectives of the Ada development include significantly reducing the number of programming languages that must be learned, supported, and maintained within the Department of Defense. The language design emphasized the goals of high program reliability, low maintenance costs, support for modern programming methodology, and efficiency of compilers and object programs [37] [38].

The language developed through competitive designs constrained by a set of requirements [13]. It is undergoing final revisions and will be frozen in mid-1980. Development of the programming environment will continue over the next two years [14]. Since compilers for the language are not yet available, it is too soon to evaluate how well the language meets its goals. However, it is possible to describe the way various features of the language are intended to respond to the abstraction issues raised here.

Although Ada grew out of the Pascal language philosophy, extensive syntactic changes and semantic extensions make it a very different language from Pascal. The major additions include module structures and interface specifications to large-program organizations and separate compilation, encapsulation facilities and generic definitions to support abstract data types, support for parallel processing, and control over low-level implementation issues related to the architecture of object machines.

There are three major abstraction tools in Ada. The **package** is used for encapsulating a set of

```
package Employee is
    restricted type PrivStuff is private;
    type EmpRec is
        record
            Name: string(1..24);
            Phone: integer;
            PrivPart: PrivStuff;
        end record;
    procedure SetSalary(Who: in out EmpRec; Sal: float);
    function GetSalary(Who: EmpRec) return float;
    procedure SetDiv(Who: in out EmpRec; Div: string(1..8));
    function GetDiv(Who: EmpRec) return string(1..8);
private
    type PrivStuff is
        record
            Salary: float;
            Division: string(1..8);
        end record;
end Employee;
```

Figure 4-5: Ada Package Definition for Employee Records

related definitions and isolating them from the rest of the program. The **type** determines the values a variable (or data structure) may take on and how it can be manipulated. The **generic** definition allows many similar abstractions to be generated from a single template, as described in Section 3.3.

The incorporation of many of these ideas into Ada can be illustrated through the example of Section 4.1. The data organization of the Pascal program (Figures 4-3 and 4-4) could be carried over almost directly to the Ada program, and the result would use Ada reasonably well. However, Ada provides additional facilities that can be applied to this problem. Recall that neither the Fortran program nor the Pascal program can allow a programmer to access names, telephone numbers, and divisions without also allowing him to access private information, here illustrated by salaries. Ada programs can provide such selected access, and we will extend the previous example to do so.[5]

We now organize the program in three components: a definition of the record for each employee (Figure 4-5), declarations of the data needed by the program (Figure 4-6), and code for construction of the telephone list (Figure 4-7).

The **package** of information about employees whose specification is shown in Figure 4-5 illustrates one of Ada's major additions to our tool kit of abstraction facilities. This definition establishes *EmpRec* as a data type with a small set of privileged operations. Only the specification of

[5]This Ada program is written in the preliminary version of Ada [37]. Revisions are currently (April 1980) being made, so this program may have become invalid when this paper appears.

the package is presented here. Ada does not require the module body to accompany the specification (though it must be defined before the program can be executed); moreover, programmers are permitted to rely only on the specifications, not on the body of a package. The specification itself is divided into a visible part (everything from **package** to **private**) and a private part (from **private** to **end**). The private part is intended only to provide information for separate compilation.

```
declare
    use Employee;

    type PhoneRec is
        record
            Name: string(1..24);
            Phone: integer;
        end record;

    Staff: array (1..1000) of EmpRec;
    Phones: array (1..1000) of PhoneRec;
    StaffSize, DivSize, i, j: integer range 1..1000;
    WhichDiv: string(1..8);
    q: PhoneRec;
```

Figure 4-6: Declarations for Ada Version of Telephone List Program

Assume that the policy for using EmpRec's is that the Name and Phone fields are accessible to anyone, that it is permissible for anyone to read but not to write the Division field, and that access to the Salary field and modification of the Division field are supposed to be done only be authorized programs. Two characteristics of Ada make it possible to establish this policy. First, the scope rules prevent any portion of the program outside a package from accessing any names except the ones listed in the visible part of the specification. In the particular case of the Employee package, this means that the Salary and Division fields of an EmpRec cannot be directly read or written outside the package. Therefore the integrity of the data can be controlled by verifying that the routines that are exported from the package are correct. Presumably the routines SetSalary, GetSalary, SetDiv, and GetDiv perform reads and writes as their names suggest; they might also keep records showing who made changes and when. Second, Ada provides ways to control the visibility of each routine and variable name. As a result, unauthorized portions of the program may be discouraged from calling routines SetSalary, GetSalary, and SetDiv; at the same time, the field names of EmpRec and routine GetDiv may be freely available everywhere.[6]

[6] Alternatively, a password could be added as a parameter to the sensitive routines.

Although the field name PrivPart is exported from the Employee package along with Name and Phone, there is no danger in doing so. An auxiliary type was defined to protect the salary and division information; the declaration

restricted type PrivStuff **is private**

indicates not only that the content and organization of the data structure are hidden from the user (**private**), but also that all operations on data of type PrivStuff are forbidden except for calls on the routines exported from the package. For **restricted** types, even assignment and comparison for equality are forbidden. Naturally, the code inside the body of the Employee package may manipulate these hidden fields; the purpose of the packaging is to guarantee that *only* the code inside the package body can do so.

```
-- Get data for division WhichDiv only

    DivSize := 0;
    for i in 1..StaffSize loop
        if GetDiv(Staff(i)) = WhichDiv then
            DivSize := DivSize + 1;
            Phones(DivSize) := (Staff(i).Name, Staff(i).Phone);
        end if;
    end loop;

-- Sort telephone list

    for i in 1..DivSize loop
        for j in i+1..DivSize loop
            if Phones(i).Name > Phones(j).Name then
                q := Phones(i);
                Phones(i) := Phones(j);
                Phones(j) := q;
            end if;
        end loop;
    end loop;
```

Figure 4-7: Code for Ada Version of Telephone List Program

The ability to force manipulation of a data structure to be carried out only through a known set of routines is central to the support of abstract data types. It is useful not only in examples such as the one given here, but also for cases in which the representation may change radically from time to time and for cases in which some kind of internal consistency among fields, such as checksums, must be maintained. Support for *secure* computation is not among Ada's goals. It can be achieved in this case, but only through a combination of an extra level of packaging and some management control. Even without guarantees about security, however, the packaging of information about how employee data is handled provides a useful structure for the development and maintenance of the program.

The declarations of Figure 4-6 are much like the declarations of the Pascal program. The `Employee` package is used instead of a simple record, and there are minor syntactic differences between the languages. The clause

```
use Employee;
```

says that all the visible names of the `Employee` package are available in the current block. (The names of the routines for manipulating `Salary` and changing `Division` must be hidden at a different point in the program.) Since Ada, unlike Pascal, allows nonprimitive types to take parameters, `Name`'s and `Division`'s are declared as `String`'s of specified length.

In the code of the Ada program itself (Figure 4-7), we assume that visibility rules allow the non-**private** field names of `EmpRecs` and the `GetDiv` function to be used. Ada provides a way to create a complete record value and assign it with a single statement; thus the assignment

```
Phones(DivSize) := (Staff(i).Name, Staff(i).Phone);
```

sets both fields of the `PhoneRec` at once. Aside from this and minor syntactic distinctions, this program fragment is very much like to the Pascal fragment of Figure 4-4.

5. Status and Potential

It is clear that methodologies and analytic techniques based on the principle of abstraction have played a major role in the development of software engineering and that they will continue to do so. In this section we describe the ways our current programming habits are changing to respond to those ideas. We also note some of the limitations of current techniques and how future work may deal with them, and we conclude with some suggestions for further reading on abstraction techniques.

5.1. How New Ideas Affect Programming

As techniques such as abstract data types have emerged, they have affected both the overall organization of programs and the style of writing small segments of code.

The new languages will have the most sweeping effects on the techniques we use for the high-level organization of program systems, and hence on the management of design and implementation projects. Modularization features that impose controls on the distribution of variable, routine, and type names can profoundly shape the strategies for decomposing a program into modules. Further, the availability of precise (and enforceable) specifications for module interfaces will influence management of software projects [76]. For example, the requirements document for a large avionics system has already been converted to a precise, if informal, specification [31]. Project organization will also be influenced by the growing availability of support tools for managing multiple modules in multiple versions [68].

The organization and style of the code within modules will also be affected. Section 4 shows how the treatment of both control and data changes within a module as the same problem is solved in languages with increasingly powerful abstraction techniques.

The ideas behind the abstract data type methodology are still not entirely validated. Projects using various portions of the methodology -- such as design based on data types, but no formal specification, or conversely specification and verification without modularity -- have been successful, but a complete demonstration on a large project has not yet been completed [63]. Although complete validation experiments have not been done, some of the initial trials are encouraging. A large, interesting program using data-type organization in a language without encapsulation facilities has been written and largely verified [21], and abstract data types specified via algebraic axioms have proved useful as a design tool [30].

5.2. Limitations of Current Abstraction Techniques

Efforts to use abstract data types have also revealed some limitations of the technique. In some cases problems are not comfortably cast as data types, or the necessary functionality is not readily expressed using the specification techniques now available. In other cases, the problem requires a set of definitions that are clearly very similar but cannot be expressed by systematic instantiation or invocation of a data type definition, even using generic definitions.

A number of familiar, well-structured program organizations do not fit well into precisely the abstract data type paradigm. These include, for example, filters and shells in the Unix spirit [42] and interactive programs in which the command syntax dominates the specification. These organizations are unquestionably useful and potentially as well-understood as abstract data types, and there is every reason to believe that similarly precise formal models can be developed. Some of these alternative points of view are already represented in high-level design systems for software [25] [57].

Although facilities for defining routines and modules whose parameters may be generic (i.e., of types that cannot be manipulated in the language) have been developed over the past five years, there has been little exploration of the *generality of generic definitions*. Part of the problem has been lack of facilities for specifying the precise dependence of the definition on its generic parameters. A specific example of a complex generic definition, giving an algorithmic transformation that can be applied to a wide variety of problems, has been written and verified [2].

The language investigations described above, together with other research projects [21] [28] [30] [33] [45] [56], have addressed questions of functional specification in considerable detail. That is, they provide formal notations such as input-output predicates, abstract models, and algebraic axioms for making assertions about the effects that operators have on program values. In many cases, the

specifications of a system cannot be reduced to formal assertions; in these cases we resort to testing in order to increase our confidence in the program [25]. In other situations, moreover, a programmer is concerned with properties other than pure functional correctness. Such properties include time and space requirements, memory access patterns, reliability, synchronization, and process independence; these have not been addressed by the data type research. A specification methodology that addresses these properties must have two important characteristics. First, it must be possible for the programmer to make and verify assertions about the properties rather than simply analyzing the program text to derive exact values or complete specifications. This is analogous to our approach to functional specifications -- we don't attempt to formally derive the mathematical function defined by a program; rather, we specify certain properties of the computation that are important and must be preserved. Further, it is important to avoid adding a new conceptual framework for each new class of properties. This implies that mechanisms for dealing with new properties should be compatible with the mechanisms already used for functional correctness.

A certain amount of work on formal specifications and verification of extra-functional properties has already been done. Most of it is directed at specific properties rather than at techniques that can be applied to a variety of properties; the results are, nonetheless, interesting. The need to address a variety of requirements in practical real-time systems was vividly demonstrated at the conference on Specifications of Reliable Software [66], most notably by Heninger [31]. Other work includes specifications of security properties [50], [18], [69], reliability [70], performance [59] [64], and communication protocols [24].

5.3. Further Reading

This paper has included extensive citations in order to make further information about briefly-discussed topics easy to obtain. The purpose of this section is to identify the books and papers that will be most helpful for general or background reading.

General issues of software development, including both management and implementation issues, are discussed in Brook's very readable book [5]. The philosophy of structured programming and the principles of data organization that underlie the representation issues of abstract data types receive careful technical treatment in [10]. The proceedings of the conference on Specifications of Reliable Software [66] contain papers on both prose descriptions of requirements and mathematical specification of abstractions.

More specific (and more deeply technical) readings include Parnas' seminal paper on information hiding [56], Guttag and Horning's discussion of the use of algebraic axioms as a design tool [30], London's survey of verification techniques [47], and papers on specification techniques including algebraic axioms [27] and abstract models [74].

6. References

1. Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, Robert E. Wells. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." *ACM SIGPLAN Notices 12*, 3 (March 1977).

2. Jon Louis Bentley and Mary Shaw. An Alphard Specification of a Correct and Efficient Transformation on Data Structures. Proceedings of IEEE Conference on Specifications of Reliable Software, IEEE, April, 1979, pp. 222-237.

3. K. L. Bowles. *Microcomputer Problem Solving Using Pascal.* Springer-Verlag, 1977.

4. Per Brinch Hansen. "The Programming Language Concurrent Pascal." *IEEE Transactions on Software Engineering SE-1* (June 1975).

5. F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, Massachusetts, 1975.

6. J. C. Browne. "The Interaction of Operating Systems and Software Engineering." *Proceedings of the IEEE 68*, 9 (September 1980).

7. J. N. Buxton and B. Randell (eds). *Software Engineering Techniques.* NATO, 1970. Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969

8. O.-J. Dahl. Simula 67 Common Base Language. Norwegian Computing Center, Oslo, 1968.

9. O.-J. Dahl and C.A.R. Hoare. Hierarchical Program Structures. In *Structured Programming*, Academic Press, 1972, pp. 175-220.

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

11. Alan M. Davis and Tomlinson G. Rauscher. Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications. Proceedings of the IEEE Conference on Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 15-35. IEEE Catalog Number 79 CH1401-9C.

12. Alan J. Demers and James E. Donahue. Data Types, Parameters and Type Checking. Proceedings of the ACM Symposium on Principles of Programming Languages, ACM SIGACT and SIGPLAN, January, 1980, pp. 12-23.

13. Department of Defense. *Steelman Requirements for High Order Computer Programming Languages.* 1978.

14. Department of Defense. *Requirements for Ada Programming Support Environments: Stoneman.* 1980.

15. Frank DeRemer and Hans H. Kron. "Programming-in-the-Large vs. Programming-in-the-Small." *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976).

16. Edsger W. Dijkstra. "Goto Statement Considered Harmful." *Communications of the ACM 11*, 3 (March 1968).

17. Edsger W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, Academic Press, 1972, pp. 1-82.

18. R. Feiertag and P.G. Neumann. The foundations of a provably secure operating system (PSOS). Proceedings of the National Computer Conference, 1979, pp. 329-334.

19. R. W. Floyd. Assigning Meanings to Programs. Proceedings of the Symposium in Applied Mathematics, American Mathematical Society, 1967, pp. 19-32.

20. Susan Gerhart and Lawrence Yelowitz. "Observations of Fallibility in Applications of Modern Programming Methodologies." *IEEE Transactions on Software Engineering SE-2*, 5 (September 1976).

21. Susan L. Gerhart and David S. Wile. Preliminary Report on the Delta Experiment: Specification and Verification of a Multiple-User File Updating Module. Proceedings of the IEEE Conference on Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 198-211. IEEE Catalog Number 79 CH1401-9C.

22. Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. "Early Experience with Mesa." *Communications of the ACM 20*, 8 (August 1977).

23. J. Goldberg. Proceedings of the Symposium on the High Cost of Software. Stanford Research Institute, September, 1973.

24. Donald I. Good. "Constructing Verified and Reliable Communications Processing Systems." *ACM Software Engineering Notes 2*, 5 (October 1977).

25. John B. Goodenough and Clement L. McGowan. "Software Quality Assurance: Testing and Validation." *Proceedings of the IEEE 68*, 9 (September 1980).

26. Loretta R. Guarino. The Evolution of Abstraction in Programming Languages. Tech. Rept. CMU-CS-78-120, Carnegie-Mellon University, May, 1978.

27. John V. Guttag. "Abstract Data Types and the Development of Data Structures." *Communications of the ACM 20*, 6 (June 1977).

28. John V. Guttag, Ellis Horowitz and David R. Musser. "Abstract Data Types and Software Validation." *Communications of the ACM 21*, 12 (December 1978).

29. John V. Guttag. "Notes on Type Abstraction (Version 2)." *IEEE Transactions on Software Engineering SE-6*, 1 (Janurary 1980), 13-23.

30. John Guttag and J.J. Horning. Formal Specification As a Design Tool. Proceedings of the ACM Symposium on Principles of Programming Languages, ACM SIGACT and SIGPLAN, January, 1980, pp. 251-261.

31. Kathryn L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Applications. Proceedings of the IEEE Conference on Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 1-14. IEEE Catalog Number 79 CH1401-9C.

32. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." *Communications of the ACM 12* (October 1969).

33. C. A. R. Hoare. "Proof of Correctness of Data Representations." *Acta Informatica 1*, 4 (1972).

34. C. A. R. Hoare. Notes on Data Structuring. In *Structured Programming*, Academic Press, 1972, pp. 83-174.

35. C. A. R. Hoare and N. Wirth. "An Axiomatic Definition of the Programming Language Pascal." *Acta Informatica 2*, 4 (1973).

36. William E. Howden. An Analysis of Software Validation Techniques for Scientific Programs. Tech. Rept. DM-171-IR, University of Victoria Department of Mathematics, March, 1979.

37. J. D. Ichbiah, et al. "Preliminary ADA Reference Manual." *ACM SIGPLAN Notices 14*, 6A (June 1979).

38. J. D. Ichbiah, et al. "Rationale for the Design of the ADA Programming Language." *ACM SIGPLAN Notices 14*, 6B (June 1979).

39. *Draft Specification for the Computer Programming Language Pascal.* International Organization for Standardization, 1979. ISO/TC 97/SC 5 N.

40. Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, 1974.

41. A. K. Jones and B. H. Liskov. An Access Control Facility for Programming Languages. Massachusetts Institute of Technology Computation Structures Group and Carnegie-Mellon University, 1976.

42. B. W. Kernighan and P. J. Plauger. *Software Tools.* Addison-Wesley, 1976.

43. Donald E. Knuth. *The Art of Computer Programming.* Volume 1: *Fundamental Algorithms.* Addison-Wesley, 1973. Second edition.

44. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek. "Report on the Programming Language Euclid." *ACM SIGPLAN Notices 12*, 2 (February 1977).

45. Barbara H. Liskov and Stephen N. Zilles. "Specification Techniques for Data Abstractions." *IEEE Transactions on Software Engineering SE-1* (March 1975).

46. Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert. "Abstraction Mechanisms in CLU." *Communications of the ACM 20*, 8 (August 1977).

47. R. L. London. A View of Program Verification. Proceedings of the International Conference on Reliable Software, April, 1975, pp, 534-545.

48. R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G.J. Popek. "Proof Rules for the Programming Language Euclid." *Acta Informatica 10*, 1 (1978), 1-26.

49. Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

50. Jonathan K. Millen. "Security Kernel Validation in Practice." *Communications of the ACM 19*, 5 (May 1976).

51. J. H. Morris. Types Are Not Sets. Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 1973, pp. 120-124.

52. J. H. Morris. "Protection in Programming Languages." *Communications of the ACM 16* (January 1973).

53. Peter Naur and Brian Randell (eds). *Software Engineering.* NATO, 1969. Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968

54. David L. Parnas. Information Distribution Aspects of Design Methodology. Proceedings of IFIP Congress, IFIP, 1971, pp. 26-30. Booklet TA-3.

55. David L. Parnas. "A Technique for Software Module Specification with Examples." *Communications of the ACM 15* (May 1972).

56. David L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 15*, 12 (December 1972).

57. Lawrence Peters. "Software Design Engineering." *Proceedings of the IEEE 68*, 9 (September 1980).

58. *Workshop on Quantitative Software Models for Reliability, Complexity, and Cost: an Assessment of the State of the Art*, 1979. IEEE Catalog No. TH0067-9.

59. Lyle Harold Ramshaw. *Formalizing the Analysis of Algorithms.* Ph.D. Th., Stanford University, 1979.

60. S. A. Schuman, Ed. "Proceedings of the International Symposium on Extensible Languages." *ACM SIGPLAN Notices 6* (December 1971).

61. Stephen A. Schuman. On Generic Functions. New Directions in Algorithmic Languages -- 1975, 1976, pp. 169-192. Stephen A. Schuman (ed)

62. Mary Shaw, Wm. A. Wulf and Ralph L. London. "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators." *Communications of the ACM 20*, 8 (August 1977).

63. Mary Shaw, Gary Feldman, Robert Fitzgerald, Paul Hilfinger, Izumi Kimura, Ralph London, Jonathan Rosenberg, and Wm. A. Wulf. Validating the Utility of Abstraction Techniques. Proceedings of ACM National Conference. ACM, December, 1978, pp. 106-110.

64. Mary Shaw. A Formal System for Specifying and Verifying Program Performance. Tech. Rept. CMU-CS-79-129, Carnegie-Mellon University, June, 1979.

65. Mary Shaw and Wm. A. Wulf. "Toward Relaxing Assumptions in Languages and Their Implementations." *SIGPLAN Notices 13*, 3 (March 1980), 45-61.

66. *Proceedings of the Conference on Specifications of Reliable Software*, 5855 Naples Plaza, Suite 301, Long Beach, California 90803, 1979. IEEE Catalog No. 79 CH1401-9C.

67. T. A. Standish. *A Data Definition Facility for Programming Languages.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, 1967.

68. *Tutorial: Automated Tools for Software Engineering*, 1979. IEEE Catalog No. EHO 150-3

69. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. "Specification and Verificaton of the UCLA Security Kernel." *Communications of the ACM 23*, 2 (Feburary 1980).

70. John H. Wensley, Leslie Lamport, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. "SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control." *Proceedings of the IEEE 66*, 10 (October 1978), 1240-1255.

71. Niklaus Wirth. "Program Development by Stepwise Refinement." *Communications of the ACM 14*, 4 (April 1971).

72. Niklaus Wirth. "Modula: A Language for Modular Programming." *Software -- Practice and Experience 7*, 1 (January 1977).

73. W. Wulf and M. Shaw. "Global Variable Considered Harmful." *ACM SIGPLAN Notices 8* (February 1973).

74. Wm. A. Wulf, Ralph L. London and Mary Shaw. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering SE-2*, 4 (December 1976).

75. Wm. A. Wulf, Mary Shaw, Lawrence Flon and Paul N. Hilfinger. *Fundamental Structures of Computer Science.* Addison-Wesley, 1980. Textbook in preparation.

76. Raymond T. Yeh and Pamela Zave. "Specifying Software Requirements." *Proceedings of the IEEE 68*, 9 (September 1980).