The Modular Structure of Complex Systems

DAVID LORGE PARNAS, PAUL C. CLEMENTS, AND DAVID M. WEISS

Abstract—This paper discusses the organization of software that is inherently complex because of very many arbitrary details that must be precisely right for the software to be correct. We show how the software design technique known as information hiding, or abstraction, can be supplemented by a hierarchically structured document, which we call a module guide. The guide is intended to allow both designers and maintainers to identify easily the parts of the software that they must understand, without reading irrelevant details about other parts of the software. The paper includes an extract from a software module guide to illustrate our proposals.

Index Terms—Abstract interfaces, information hiding, modular structure of software, software engineering.

I. Introduction

MORE than five years ago, a number of people at the Naval Research Laboratory became concerned about what we preceived to be a growing gap between software engineering principles being advocated at major conferences and the practice of software engineering at many industrial and governmental laboratories. The conferences and many journals were filled with what appeared to be good ideas, illustrated using examples that were either unrealistically simple fragments or complex problems that were not worked out in much detail. When we examined actual software projects and their documentation, few showed any use of the ideas and no successful product appeared to have been designed by consistent application of the principles touted at conferences and in journals. The ideas appeared to be easier to write about than to use.

We could imagine several reasons for the gap: 1) the ideas were, as many old-style programmers claim, simply impractical for real problems; 2) responsible managers were unwilling to bet on principles that had been not been proven in practice, thus creating a startup problem; 3) the examples used in the papers were too unlike the problems of the practitioners to serve as models; 4) the ideas might need refinement or extension before they could be used as guidelines for projects with the complexity and resource constraints found in the field; and 5) the practitioners were, as some academics claim, not intellectually capable of the tasks given them. Our familiarity with both the ideas and the practitioners led us to reject 1) and 5); we decided to see what could be done about 2)–4).

Our decision was to take an undeniably realistic problem and to apply the "academic" ideas to it, so that if we succeeded, 1) there would be evidence of the feasibility for responsible managers; 2) there would be a model for use by others with similar problems; and 3) we could refine or supplement the ideas until they would work for systems more complex than those in the literature. We chose to build an exact duplicate of an existing system so that it would be possible to compare the system built by conventional techniques to one built in accordance with the new academic principles. The project chosen was the Operational Flight Program (OFP) for the A-7E aircraft. The current program uses many dirty tricks, barely fits in its memory, and barely meets its real-time constraints. Consequently, we felt that this program, although much smaller than many programs, was a realistic test of the ideas. Because the current OFP is considered one of the best programs of its ilk, we considered the task sufficiently challenging that skeptics would not attribute our success to the poor quality of the program that we are trying to match.

Although the project is far from complete, we have already had some limited success in all three of our goals. Our ability to write a complete and precise requirements specification for the software has encouraged managers to try the same approach, and our document [9] has served as a model for those projects. We have also found useful refinements of the principles that we advocated before starting the project. For example, the concept of abstract interfaces, which we discussed in [1], has now been refined and illustrated in [2] and [3].

This paper presents another refinement of the principles that we set out to use. One of the most basic ideas in our approach was the use of the principle of information hiding [6] to decompose a project into work assignments or modules. This idea was an excellent example of the gap between academic software engineering and practice. While it has been considered self-evident by some academics, we could find no sizable product in which the idea had been consistently used. While some authors were treating the idea as "old hat," we could not persuade those charged with building real software to do something so radically different from what they had been doing.

When we tried to use the idea we found that while it was quite applicable, some additional ideas were necessary to make it work for systems with more than a dozen or so modules. This paper discusses the problems that we encountered and the additional ideas.

II. Background and Guiding Principles

A. Three Important Software Structures

A structural description of a software system shows the program's decomposition into parts and the relations between those parts. A-7E programmers must be concerned with three

Manuscript received January 31, 1984.
D. L. Parnas is with the University of Victoria, Victoria, B.C., Canada, and the Computer Science and Systems Branch, U.S. Naval Research Laboratory, Washington, DC 20375.
P. C. Clements and D. M. Weiss are with the Computer Science and Systems Branch, U.S. Naval Research Laboratory, Washington, DC 20375.

structures: a) the module structure, b) the uses structure, and c) the process structure. This section contrasts these structures.

a) A module is a work assignment for a programmer or programmer team. Each module consists of a group of closely related programs. The module structure is the decomposition of the program into modules and the assumptions that the team responsible for each module is allowed to make about the other modules.

b) In the uses structure the components are programs, i.e., not modules but parts of modules; the relation is "requires the presence of." The uses structure determines the executable subsets of the software [5].

c) The process structure is a decomposition of the run-time activities of the system into units known as processes. Processes are not programs; there is no simple relation between modules and processes. The implementation of some modules may include one or more processes, and any process may invoke programs in several modules.

The rest of this paper discusses the module structure.

B. Design Principle

Our module structure is based on the decomposition criterion known as information hiding [6]. According to this principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Each data structure is used in only one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

Appying this principle is not always easy. It is an attempt to minimize the expected cost of software and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience, and may require knowledge of the application area, as well as an understanding of hardware and software technology. Because a designer may not have all of the relevant experience, we have developed formal review procedures designed to take advantage of others that do have that experience. These procedures are described in [2].

C. Goals of Modular Structure

The primary goal of the decomposition into modules is reduction of overall software cost by allowing modules to be designed and revised independently. Specific goals of the module decomposition are as follows.

a) Each module's structure should be simple enough to be understood fully.

b) It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of other modules.

c) The case of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed. It should be possible to make likely changes without changing any module interfaces; less likely changes may involve interface changes, but only for modules that are small and not widely used. Only very unlikely changes should require changes in the interfaces of widely used modules.

d) It should be possible to make a major software change as a set of independent changes to individual modules, i.e., except for interface changes, programmers changing the individual modules should not need to communicate. If the interfaces of the modules are not revised, it should be possible to run and test any combination of old and new module versions.

As a consequence of the goals above, our software is composed of many small modules. In previous attempts to use information hiding, we had seen systems with 5-20 modules. We know now that we will have hundreds of modules. With 25 or fewer modules it would not be difficult to know which modules would be affected by a change. With hundreds of modules that is not the case. With 25 or fewer modules, careful inspection may suffice to make sure that nothing has been overlooked. With hundreds of modules we found that impossible. We realized that the use of information hiding could backfire. With most maintainers ignorant about the internal structure of most of the modules, maintainers would have to search through many module documents to find the ones they had to change. We also feared working for some time before discovering that we had left out some major modules.

We concluded that we needed some additional discipline in applying the information hiding principle, and that special documentation was needed if we were really to reduce the cost of maintining complex software systems. We had to find a way to work with small lists of modules so that we could prepare convincing arguments that each list was complete. We needed to prepare a software module guide that would assist the maintenance programmer in finding the modules that were affected by a change or could be causing the problem.

As a result of these considerations, the modules have been organized into a tree-structured hierarchy; each nonterminal node in the tree represents a module that is composed of the modules represented by its descendents. The hierarchical structure has been documented in a module guide [9]. The hierarchy and the guide are intended to achieve the following additional goals.

e) A software engineer should be able to understand the responsibility of a module without understanding the module's internal design.

f) A reader with a well-defined concern should easily be able to identify the relevant modules without studying irrelevant modules. This implies that the reader be able to distinguish relevant modules from irrelevant modules without looking at their components.

g) The number of branches at each nonterminal module in the graph should be small enough that the designers can prepare convincing arguments that the submodules have no overlapping responsibilities, and that they cover all of the responsibilities that the module is intended to cover. This is most valuable during the initial design, but it also helps when identifying modules affected by a change.

D. Restricted and Hidden Modules

We found that it was not always possible to confine information to a single module in a real system. For example, information about hardware that could be replaced should be confined, but diagnostic information about that hardware must be communicated to modules that display information to users or hardware maintainers. Any program that uses that information is subject to change when the hardware changes. To reduce the cost of software changes, the use of modules that provide such information is restricted. Restricted interfaces are indicated by "(*R*)" in the Guide. Often the existence of certain smaller modules is itself a secret of a larger module. In a few cases, we have mentioned such modules in this document in order to clearly specify where certain functions are performed. Those modules are referred to as hidden modules and indicated by "(*H*)" in the documentation.

E. Module Description

The Module Guide shows how responsibilities are allocated among the major modules. Such a guide is intended to lead a reader to the module that implements a particular aspect of the system. It states the criteria used to assign a particular responsibility to a module and arranges the modules in such a way that a reader can find the information relevant to his purpose without searching through unrelated documentation. The guide defines the scope and contents of the individual design documents.

Three ways to describe a module structure based on information hiding are: 1) by the roles played by the individual modules in the overall system operation; 2) by the secrets associated with each module; and 3) by the facilities provided by each module. The module guide describes the module structure by characterizing each module's secrets. Where useful, a brief description of the role of the module is included. The detailed description of facilities for modules is relegated to other documents called "module specifications"; e.g., [2]. The module guide tells you which module(s) will require a change. The module specification tells you both how to use that module and what that module must do.

For some modules we find it useful to distinguish between a primary secret, which is hidden information that was specified to the software designer, and a secondary secret, which refers to implementation decisions made by the designer when implementing the module designed to hide the primary secret.

In the module guide we attempted to describe the decomposition rules as precisely as possible, but the possibility of future changes in technology makes some boundaries fuzzy. Where this occurs we note fuzzy areas and discuss additional information used to resolve ambiguities.

F. The Illustrative Example

To show how our techniques work, we give a fairly large extract from the module guide for the A-7 OFP. We discuss the way that it helps during construction and maintenance after the extract.

The design that we present is the module structure of the A-7E flight software produced by the Naval Research Laboratory. The A-7E flight software is a hard real-time program that processes flight data and controls displays for the pilot. It computes the aircraft position using an inertial navigation system, and must be highly accurate. The current operational program is best understood as one big module. It is very difficult to identify the sections of the program that must be changed when certain requirements change. Our software structure is designed to meet the goals mentioned above, but must still meet all accuracy and real-time constraints.

What follows is an extract from the module guide for NRL's version of the software [7]. A complete copy of the guide or any of the other NRL reports can be obtained by writing to

Code 7590
Naval Research Laboratory
Washington, DC 20375

III. A-7E Module Structure

A. Top Level Decomposition

The software system consists of the three modules described below.

A.1 Hardware-Hiding Module

The hardware-hiding module includes the programs that need to be changed if any part of the hardware is replaced by a new unit with a different hardware-software interface but with the same general capabilities. This module implements virtual hardware that is used by the rest of the software. The primary secrets of this module are the hardware-software interfaces described in chapters 1 and 2 of the requirements document [9]. The secrets of this module are the data structures and algorithms used to implement the virtual hardware.

A.2 Behavior-Hiding Module

The behavior-hiding module includes programs that need to be changed if there are changes in the sections of the requirements document that describe the required behavior [9, ch. 3, 4]. The content of those sections is the primary secret of this module. These programs determine the values to be sent to the virtual output devices provided by the hardware-hiding module.

A.3 Software Decision Module

The software decision module hides software design decisions that are based upon mathematical theorems, physical facts, and programming considerations such as algorithmic efficiency and accuracy. The secrets of this module are *not* described in the requirements document. This module differs from the other modules in that both the secrets and the interfaces are determined by software designers. Changes in these modules are more likely to the motivated by a desire to improve performance than by externally imposed changes.

Notes on the Top-Level Decomposition

Fuzziness is present in the above classifications for the following reasons.

a) The line between requirements definition and software design has been determined in part by decisions made when the requirements documents are written; for example, weapon trajectory models may be chosen by system analysts and specified in the requirements document, or they may be left to the discretion of the software designers by stating accuracy requirements but no algorithms.

b) The line between hardware characteristics and software design may vary. Hardware can be built to perform some of the services currently performed by the software; consequently, certain modules can be viewed either as modules that hide hardware characteristics or as modules that hide software design decisions.

c) Changes in the hardware or in the bahavior of the system or its users may make a software design decision less appropriate.

d) All software modules include software design decisions;

changes in any module may be motivated by efficiency or accuracy considerations.

Such fuzziness would be unacceptable for our purposes. We can eliminate it by referring to a precise requirements document such as [9]. That document specifies the lines between behavior, hardware, and software decisions.

a) When the requirements document specifies an algorithm, we do not consider the design of the algorithm to be a software design decision. If the requirements document only states requirements that the algorithm must meet, we consider the program that implements that algorithm to be part of a software decision module.

b) The interface between the software and the hardware is specified in the software requirements document. The line between hardware characteristics and software design must be based on estimates of the likelihood of future changes. If it is reasonably likely that future hardware will implement a particular facility, the software module that implements that facility is classified as a hardware-hiding module; otherwise, the module is considered a software design module. We have consistently taken a conservative stance; the design is based on the assumption that drastic changes are less likely than evolutionary changes. If there are changes to the aspects of the hardware hiding module. If there are radical changes that provide services previously provided by software, some of the software decision modules may be eliminated or reduced in size.

c) A module is included in the software decision module only if it would remain useful, although possibly less efficient, when there are changes in the requirements document.

d) A module will be included in the software decision category only if its secrets do not include information documented in the software requirements document.

B. Second-Level Decomposition

B.1 Hardware-Hiding Module Decomposition

The hardware hiding module comprises two modules.

B.1.1 Extended Computer Module

The extended computer module hides those characteristics of the hardware–software interface of the avionics computer that we consider likely to change if the computer is modified or replaced.

Avionics computers differ greatly in their hardware–software interfaces and in the capabilities that are implemented directly in the hardware. For example, some avionics computers include a floating-point approximation of real numbers, while others perform approxiamte real number operations by a programmed sequence of fixed-point operations. Some avionics systems include a single processor; some systems provide several processors. The extended computer provides an instruction set that can be implemented efficiently on most avionics computers. This instruction set includes the operations on application-independent data types, sequence control operations, and general I/O operations.

The primary secrets of the extended computer are: the number of processors, the instruction set of the computer, and the computer's capacity for performing concurrent operations.

The structure of the extended computer module is given in Section C.1.1.

B.1.2 Device Interface Module

The device interface module hides those characteristics of the peripheral devices that are considered likely to change. Each device might be replaced by an improved device capable of accomplishing the same tasks. Replacement devices differ widely in their hardware–software interfaces. For example, all angle-of-attack sensors measure the angle between a reference line on the aircraft and the velocity of the surrounding air mass, but they differ in input format, timing, and the amount of noise in the data.

The device interface module provides virtual devices to be used by the rest of the software. The virtual devices do not necessarily correspond to physical devices, because all of the hardware providing a capability is not necessarily in one physical unit. Furthermore, there are some capabilities of a physical unit that are likely to change independently of others; it is advantageous to hide characteristics that may change independently in different modules.

The primary secrets of the device interface module are those characteristics of the present devices documented in the requirements document and not likely to be shared by replacement devices.

The structure of the device interface module is given in Section C.1.2.

Notes on the Hardware-Hiding Module Decomposition

Parts of the hardware were considered external devices by those who designed the CPU but are treated as part of the processor by other documents. Our distinction between computer and device is based on the current hardware and is described in the requirements document. Information that applies to more than one device is considered a secret of the extended computer; information that is only relevant to one device is a secret of a device interface module. For example, there is an analog-to-digital converter that is used for communicating with several devices; it is hidden by the extended computer, although it could be viewed as an external device. As another example, there are special outputs for testing the I/O channels; they are not associated with a single device. These are the responsibility of the extended computer.

If all the hardware were replaced simultaneously, there might be a significant shift in responsibilities between computer and devices. In systems like the A-7E, such changes are unusual; the replacement of individual devices or the replacement of the computer alone is more likely. Our design is based on the expectation that this pattern of replacement will continue to hold.

B.2 Behavior-Hiding Module Decomposition

The behavior hiding module consists of two modules: a function driver (FD) module supported by a shared services (SS) module.

B.2.1 Function Driver Module

The function driver module consists of a set of individual modules called function drivers; each function driver is the sole controller of a set of closely related outputs. Outputs are considered closely related if it is easier to describe their values together than individually. For example, if one output is the sine of an angle, the other the cosine of the same angle, a joint description of the two will be smaller than two separate descriptions. Note that the function driver modules deal with outputs to the virtual devices created by the hardware hiding modules, not the physical outputs. The primary secrets of the function driver module are the rules determining the values of these outputs.

The structure of the function driver module is given in Section C.2.1.

B.2.2 Shared Services Module

Because all the function drivers control systems in the same aircraft, some aspects of the behavior are common to several function drivers. We expect that if there is a change in that aspect of the behavior, it will affect all of the functions that share it. Consequently, we have identified a set of modules, each of which hides an aspect of the behavior that applies to two or more of the outputs.

The structure of the shared services module is found in Section C.2.2.

Notes on Behavior-Hiding Module Decomposition

Because users of the documentation cannot be expected to know which aspects of a function's behavior are shared, the documentation for the function driver modules will include a reference to the shared services modules that it uses. A maintenance programmer should always begin his inquiry with the appropriate function driver. He will be directed to the shared services modules when appropriate.

B.3 Software Decision Module Decomposition

The software decision module has been divided into (1) the application data type module, which hides the implementation of certain variables, 2) the physical model module, which hides algorithms that simulate physical phenomena, 3) the data banker module, which hides the data-updating policies, 4) the system generation module, which hides decisions that are postponed until system generation time, and 5) the software utility module, which hides algorithms that are used in several other modules.

B.3.1 Application Data Type Module

The application data type module supplements the data types provided by the extended computer module with data types that are useful for avionics applications and do not require a computer dependent implementation. These data types are implemented using the data types provided by the extended computer; variables of those types are used just as if the types were built into the extended computer.

The secrets of the application data type module are the data representation used in the variables and the programs used to implement operations on those variables. These variables can be used without consideration of units. Where necessary, the modules provide conversion operators, which deliver or accept real values in specified units.

Run-time efficiency considerations sometimes dictate that an implementation of an application data type be based on a secret of another module. In that case, the data type will be specified in the application data type module documentation, but the implementation will be described in the documentation, but the implementation will be described in the documentation will contain the appropriate references in such cases.

The structure of the application data type module is given in Section C.3.1.

B.3.2 Physical Model Module

The software requires estimates of quantities that cannot be measured directly but can be computed from observables using models of the physical world. The primary secrets of the physical model module are the physical models; the secondary secrets are the computer implementations of those models.

The structure of the physical model module is given in Section C.3.2.

B.3.3 Data Banker Module

Most data are produced by one module and "consumed" by another. Usually the consumers should receive a value as up-to-date as practical. The data banker module acts as a "middleman" and determines when new values for these data are computed. The data banker obtains values from producers; consumer programs obtain data from data banker access programs. The producer and consumers of a particular datum can be written without knowing whether or not the data banker stores the value or when a stored value is updated. In most cases, neither the producer nor the consumer need be modified if the updating policy changes.

The data banker is not used if consumers require specific members of the sequence of values computed by the producer, or if they require values associated with a specific time, such as the moment when an event occurs.

Some of the updating policies that can be implemented in the data banker are described in the following table, which indicates whether or not the data banker stores a copy of the item and when a new value is computed.

Name	*Storage*	*When new value produced*
on demand:	No	Whenever a consumer requests the value
periodic:	Yes	Periodically. Consumers get the most recently stored value.
event driven:	Yes	Whenever the data banker is notified, by the occurrence of an event, that the value may have changed. Consumers get the most recently stored value.
conditional:	Yes	Whenever a consumer requests the value, provided certain conditions are true. Otherwise, a previously stored value is delivered.

The choice among these and other updating policies should be based on the consumers' accuracy requirements, how often consumers require the value, the maximum wait that consumers can accept, how often the value changes, and the cost of producing a new value. Since the decision is not based on coding details of either consumer or producer, it is usually not necessary to rewrite a data banker module when producer or consumer change.

B.3.4 System Generation Module

The primary secrets of the system generation module are decisions that are postponed until system-generation time. These include the values of system generation parameters and the choice among alternative implementations of a module. The secondary secrets of the system generation module are the method used to generate a machine-executable form of the code and the representation of the postponed decisions. Most of the programs in this module do not run on the on-board computer; they run on a more powerful computer used to generate the code for the on-board system. Some of the programs are tools provided with our system; others have been developed specifically for this project.

B.3.5 Software Utility Module

The primary secrets of this module are the algorithms implementing common software functions such as resource monitor modules, and mathematical routines such as square-root and logarithm.

C. Third-Level Deocmposition

Note: For the purposes of this paper, only third-level modules whose descriptions are particularly illustrative are included. Ellipses indicate omissions.

C.1 Extended Computer Module Decomposition

C.1.1.1 Data Type Module

The data type module implements variables and operators for real numbers, time periods, and bit strings. The data representations and data manipulation instructions built into the computer hardware are the primary secrets of this module—specifically, the representation of numeric objects in terms of hardware data types; the representation of bitstrings; how to access a bit within a bitstring; and how times are represented for the hardware timers. The secondary secrets of this module are how range and resolution requirements are used to determine representation; the procedures for performing numeric operations; the procedures used to perform bitstring operations; and how to compute the memory location of an array element given the array name and the element index.

. . .

C.1.1.4 Computer State Module

The computer state module keeps track of the current state of the extended computer, which can be either operating, off, or failed, and signals relevant state changes to user programs. The primary secret is the way that the hardware detects and causes state changes. After the EC has been initialized, this module signals the event that starts the initialization for the rest of the software.

. . .

C.1.1.7 Diagnostics Module (R)

The diagnostics module provides diagnostic programs to test the interrupt hardware, the I/O hardware, and the memory. Use of this module is restricted because the information it returns reveals secrets of the extended computer, i.e., programs that use it may have to be revised if the avionics computer is replaced by another computer.

C.1.1.8 Virtual Memory Module (H)

The virtual memory module presents a uniformly addressable virtual memory for use by DATA, I/O, and SEQUENCE submodules, allowing them to use virtual addresses for both data and subprograms. The primary secrets of the virtual memory module are the hardware addressing methods for data and instructions in real memory; differences in the way that different areas of memory are addressed are hidden. The secondary secrets of the module are the policy for allocating real memory to virtual addresses and the programs that translate from virtual address references to real instruction sequences.

. . .

C.1.2 Device Interface Module Decomposition

The following table describes the device interface submodules (DIM's) and their secrets. The phrase "how to read . . ." is intended to be interpreted quite liberally, e.g., it includes device-dependent corrections, filtering, and any other actions that may be necessary to determine the physical value from the device input. All of the DIM's hide the procedures for testing the device that they control.

Section	*Virtual Device*	*Secret: How to . . .*
C.1.2.1	Air data computer	read barometric altitude, true airspeed, and Mach number.
C.1.2.2	Angle of attack sensor	read angle of attack.
. . .		
C.1.2.20	Weapon release system	ascertain weapon release actions the pilot has requested; cause weapons to be prepared and released.

. . .

C.2.1 Funtion Driver Module Decomposition

The following table describes the function driver submodules and their secrets.

Section	*Function Driver*	*Secret*
. . .		
C.2.1.7	Head-up display functions	Where the movable HUD symbols should be placed. Whether a HUD symbol should be on, off, or blinking. What information should be displayed on the fixed-position displays.
C.2.1.8	Inertial measurement set functions	Rules determining the scale to be used for the IMS velocity measurements. When to initialize the velocity measurements. How much to rotate the IMS for alignment.

C.2.1.9	Panel functions	What information should be displayed on panel windows. When the enter light should be turned on.

. . .

C.2.2 Shared Services Module Decomposition

The shared services module comprises the following modules.

C.2.2.1 Mode Determination Module

The mode determination module determines system modes (as defined in the requirements document). It signals the occurrence of mode transitions and makes the identity of the current modes available. The primary secrets of the mode determination module are the mode transition tables in the requirements document.

C.2.2.4 System Value Module

A system value submodule computes a set of values, some of which are used by more than one function driver. The secrets of a system value submodule are the rules in the requirements that define the values that it computes. The shared rules in the requirements specify such things as 1) selection among several alternative sources, 2) applying filters to values produced by other modules, or 3) imposing limits on a value calculated elsewhere.

This module may include a value that is only used in one function driver if the rule used to calculate that value is the same as that used to calculate other shared values.

Each system value submodule is also responsible for signaling events that are defined in terms of the values it computes.

. . .

C.3.1 Application Data Type Module Decomposition

The application data type module is divided into two submodules.

C.3.1.1 System Data Type Module

The system data type module implements variables of the following widely used types: accelerations, angles, angular rates, character literals, densities, Mach values, distances, pressures, and speeds. These modules may be used to implement types with restricted ranges or special interpretations (e.g., angle is used to represent latitude).

C.3.1.2 State Transition Event Module

The STE module implements variables that are instances of finite state machines. Users can await the transition of a variable to/from a particular state value, cause transitions, and compare values for equality.

C.3.2 Physical Model Module Decomposition

The physical model module comprises the modules described below.

C.3.2.1 Earth Model Module

The earth model module hides models of the earth and its atmosphere. This set of models includes models of local gravity, the curvature of the earth, pressure at sea level, magnetic variation, the local terrain, and rotation of the earth, coriolis force, and atmospheric density.

C.3.2.2 Aircraft Motion Module

The Aircraft motion module hides models of the aircraft's motion. They are used to calculate aircraft position, velocity, and attitude from observable inputs.

C.3.2.3 Spatial Relations Module

The spatial relations module contains models of three-dimensional space. These models are used to perform coordinate transformations as well as angle and distance calculations.

C.3.2.4 Human Factors Module

The human factors module is based on models of pilot reaction time and perception of simulated continuous motion. The models determine the update frequency appropriate for symbols on a display.

C.3.2.5 Weapon Behavior Module

The weapon behavior module contains models used to predict weapon behavior after release.

. . .

IV. Conclusions

Any conclusions that we draw at this point must be considered tentative, as they have not been confirmed by the production of a running program. Nonetheless, we have been using the module guide for several years and it has proven remarkably stable. It plays a significant role in our development process; programmers and designers turn to it when they are unsure about where a certain program should reside. Numerous discussions have been resolved by this means, and relatively few and superficial changes have resulted from the discussions.

Our experience suggests that the use of information hiding in complex systems is practical, but only if the design begins with the writing of a module guide that is used to guide the design of the individual module interfaces. When we tried to work without the guide, numerous problems slipped between the cracks and responsibilities ended up either in two modules or in none. With the module guide, further progress on the design has revealed relatively few oversights. New programmers joining the project are able to get a quick grasp of the structure of our project without using much time talking to those who have been on the project longer. We feel that this will help to ameliorate Brooks' adage, "Adding more men then lengthens, not shortens, the schedule" [8].

We realize that the module guide that we are using as an illustration stops at an arbitrary point. Most of the modules mentioned in this guide are divided into submodules that are not shown in this guide. We found it more convenient to have separate module guides for the smaller modules than to keep extending this one. This module guide is the one document that all implementors must read; the others are for specialists. This one is less than 30 pages in length and we can afford to let everyone read it.

In writing this and other module guides, we have seen how important it is to focus on describing secrets rather than interfaces or roles of the modules. Where we have forgotten that (usually when we are rushing to meet a dealine), we have ended up with modules without clear responsibilities and eventually had to revise our design.

The Module Guide, like our requirements document, provides a clear illustration of the advantages of an approach that we call "design through documentation" [4]. Writing the document is our way of making progress in design. The document then serves to guide us and others in future designs.

In another paper [10], we have argued that this approach increases the likelihood that the software we produce will be reusable and reused. That paper uses the same example to argue rather different points.

ACKNOWLEDGMENT

K. Britton, now with IBM, Research Triangle Park, NC, is a coauthor of our software module guide. Parts of that guide have been included in this paper.

REFERENCES

[1] D. Parnas, "Use of abstract interfaces in the development of software for embedded computer systems," Naval Res. Lab., Washington, DC, NRL Rep. 8047, June 1977.

[2] A. Parker, K. Heninger, D. Parnas, and J. Shore, "Abstract interface specifications for the A-7E device interface module," Naval Res. Lab., Washington, DC. NRL Memo. Rep. 4385, Nov. 20, 1980.

[3] K. Britton, A. Parker, and D. Parnas, "A procedure for designing abstract interfaces for device interface modules," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981.

[4] S. Hester, D. Parnas, and D. Utter, "Using documentation as a software design medium," *Bell Syst. Tech. J.*, vol. 60, pp. 1941–1977, Oct. 1981.

[5] D. Parnas, "Designing software for ease of extension and contraction," in *Proc. 3rd Int. Conf. Software Eng.*, May 1978; see also *IEEE Trans. Software Eng.*, vol. SE-5, Mar. 1979.

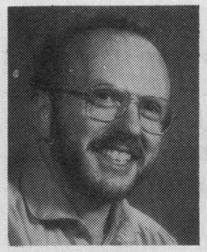
[6] —, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, Dec. 1972.

[7] K. Britton and D. Parnas, "A-7E software module guide," Naval Res. Lab., Washinton, DC, NRL Memo. Rep. 4702, Dec. 1981.

[8] F. P. Brooks, Jr., *The Mythical Man Month–Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

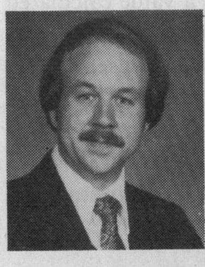
[9] K. Heninger, J. Kallander, D. Parnas, and J. Shore, "Software requirements for the A-7E aircraft," Naval Res. Lab., Washington, DC, NRL Memo. Rep. 3876, Nov. 27, 1978.

[10] P. Clements, D. Parnas, and D. Weiss, "Enhancing reuseability with information coding," in *Proc. Workshop Reuseability in Programming*, Sept. 1983.



David Lorge Parnas was born in Plattsburgh, NY, on February 10, 1941.

He is currently Lansdowne Professor of Computer Science at the University of Victoria, Victoria, B.C., Canada, as well as Principle Investigator of the Software Cost Reduction Project at the Naval Research Laboratory, Washington, DC. He has also taught at Carnegie-Mellon University, the University of Maryland, the Technische Hochschule Darmstadt, and the University of North Carolina at Chapel Hill. He is interested in all aspects of software engineering. His special interests include program semantics, language design, program organization, process structure, process synchronization, and precise abstract specifications. He is currently leading an experimental redesign of a hard-real-time system in order to evaluate a number of software engineering principles. He is also involved in the design of a language involving new control structures and abstract data types.



Paul C. Clements received the B.S. degree in mathematical sciences in 1977 and the M.S. degree in computer science in 1980, both from the University of North Carolina at Chapel Hill.

Since 1980 he has worked in software engineering research at the Naval Research Laboratory, Washington, DC. In 1982 he became the Technical Coordinator of the Software Cost Reduction Project, whose purpose is to provide a well-engineered model of a complex real-time system.

He is interested in most areas of software engineering, but most of his time is spent working on problems in modularization and specification of software designs.

David M. Weiss, for a photograph and biography, see p. 168 of the February 1985 issue of this TRANSACTIONS.