

ESD ACCESSION LIST,

DRI Call No. 78230

Copy No. 1 of 2 cys.

ESD-TR-72-164

MTR-2345

GUIDELINES FOR THE DESIGN AND IMPLEMENTATION
OF RELIABLE SOFTWARE SYSTEMS

B. H. Liskov

FEBRUARY 1973

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts



ESD TR-72-164
FEB 1973
SCIENTIFIC & TECHNICAL ASSISTANT DIVISION
HANSCOM FIELD

Approved for public release;
distribution unlimited.

Project 671A

Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-71-C-0002

AD757905

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

GUIDELINES FOR THE DESIGN AND IMPLEMENTATION
OF RELIABLE SOFTWARE SYSTEMS

B. H. Liskov

FEBRUARY 1973

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts



Approved for public release;
distribution unlimited.

Project 671A

Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts

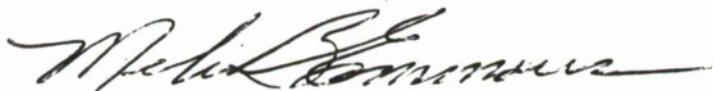
Contract No. F19628-71-C-0002

FOREWORD

The work described in this report was carried out under the sponsorship of the Deputy for Command and Management Systems, Project 671A, by The MITRE Corporation, Bedford, Massachusetts, under Contract No. F19628-71-C-0002.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.



MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command and Management Systems

ABSTRACT

This document describes experimental guidelines governing the production of reliable software systems. Both programming and management guidelines are proposed. The programming guidelines are intended to enable programmers to cope with a complex system effectively. The management guidelines describe an organization of personnel intended to enhance the effect of the programming guidelines.

PREFACE

This report has been produced under the Highly Reliable Programming Task in support of Air Force Project 5550 of the Advanced Development Program. The necessity for reliable software in computer-based Air Force systems such as AABNCP is easily recognized; current methods of system development have seldom resulted in software which is free of errors. Therefore, the objectives of this task have been to determine which available techniques might facilitate production of reliable software, to demonstrate how these techniques can be applied to software development, and finally to recommend to the Air Force ways to realize the benefits of these techniques.

The first step taken in this task was to perform a survey of the literature and an analysis of current approaches to the problem of software reliability.⁽¹⁾ One of these approaches, called the "constructive" approach, was selected as the most feasible at this time. This approach is concerned with a methodology for system development which seeks to eliminate the sources of errors by making a concern for reliability an integral part of the development process.

This report represents the second step in meeting the objectives of the Highly Reliable Programming Task. In order to demonstrate the effectiveness of the constructive approach, a set of guidelines has been developed. These guidelines govern the application of a combination of techniques which separately have been used to aid in the production of reliable software.

The guidelines described here will be conscientiously applied to the construction of a small, but complex, file management system so that they may be evaluated and refined. Ultimately the guidelines will be restated in a form suitable for use by the Air Force in controlling the development of software systems, either in-house or acquired from contractors, so that the resulting software will be reliable.

TABLE OF CONTENTS

		<u>Page</u>
SECTION I	INTRODUCTION	1
	OBJECTIVES	1
	DESCRIPTION OF PAPER	2
SECTION II	BASIC TECHNIQUES	3
	MODULARITY	3
	STRUCTURED PROGRAMMING	5
	Structured Programs and Higher Level Languages	6
	Structured Programming and Proofs of Correctness	6
	Structured Programs and Levels of Abstraction	7
	Structured Programming and System Design	7
SECTION III	PROGRAMMING GUIDELINES	8
	GUIDELINES FOR SELECTING LEVELS OF ABSTRACTION	9
	Useful Abstractions	9
	Simplification of Levels of Abstraction	10
	System Maintenance and Modification	11
	A Special Guideline for Asynchronous Systems	11
	An Example from Venus	12
SECTION IV	GUIDELINES FOR BUILDING SYSTEMS	15
	THE DESIGN PHASE	16
	How to Proceed with the Design	17
	Design of an Asynchronous System	17
	Special Problems Which Must Be Solved During the Design	18
	Structured Programming	19
	Specifications of Levels and Functions	19
	End of the Design Phase	20
	Organization of Personnel	21
	Design Meetings and Project Documentation	22
	IMPLEMENTATION AND TESTING	23
	Design Within Implementation	24
	Order of Implementation	24
	Specifications	25
	Order of Testing	25
	Organization of Personnel	25
	The Programming Secretary	26
Project Documentation	27	
Documentation of the System	29	
REFERENCES		30

SECTION I

INTRODUCTION

OBJECTIVES

The purpose of this paper is to establish experimental guidelines governing the building of reliable software systems. These guidelines will be tested by attempting to follow them when building an on-line interactive multi-user file management system later this year. Throughout the building of the system, the success or failure of the guidelines will be noted, hopefully leading to the establishment of a set of more useable guidelines as a result.

Our primary objective is to learn how to build reliable software, containing no errors upon delivery. It is customary to divide the building of a software system into three phases: design, implementation and testing. Design involves both making decisions about what precisely a system does and then planning an overall structure for the software which permits it to perform its tasks. This structure is traditionally represented by flowcharts. Implementation consists of writing the programs which make up the software system; these programs fit into the structure specified by the design. Testing is debugging of the software; it is usually performed first on the individual programs and then on combinations of programs (system integration).

In the title of this paper, only design and implementation are mentioned. This is done purposefully to underline the approach of this project, which considers testing as an integral part of design and implementation. The standard approach to building software systems, involving extensive debugging, has not proved successful in practice. As Dijkstra said, "Testing shows the presence, not the absence of bugs."⁽²⁾ What is required is a methodology for designing and implementing systems which permits (informal) proofs of the correctness of the software before testing begins. These proofs will turn up relevant test cases. Another requirement on the software is that there is a small number of such test cases; this is only possible if considerations about testing influenced design and implementation. If this is true, then it will be possible to exhaustively check every test case. When this exhaustive testing is combined with the informal proofs, it is reasonable to expect software reliability after testing is complete. This expectation is borne out by certain experiments performed in the past.^(3,4)

In addition to producing reliable software, it is also necessary to produce readable software which is relatively easy to modify and maintain. Systems of any size can always be expected to be subject to changes in requirements, resulting in recoding of parts of the system. A reliable system which cannot adjust to such changes is therefore not satisfactory.

DESCRIPTION OF PAPER

We will consider building complex software systems. A two-fold definition is offered for "complex." First, there are many system states in such a system, and it is difficult to organize the program logic to handle all states correctly; programming guidelines will be proposed whose purpose is to help programmers deal with this complexity effectively. (In this paper, all people concerned with production of software, both in design and implementation, are called programmers.) Second, the efforts of many individuals must be coordinated in order to build the system; management guidelines will be proposed governing the organization of the programmers, the coordination of their efforts, the communication between them, and the scheduling of the project as a whole. In addition, management guidelines provide support for programming guidelines. It is generally accepted that the organizational structure of people imposes a structure on the system being built.⁽⁵⁾ Since we want a certain system structure as established by the programming guidelines, we must therefore structure the people correctly in order to achieve this.

The guidelines proposed in this paper are experimental, and it is too much to expect that they are complete or even correct. It is fairly easy to look at a given system and say that it is poorly constructed. It is more difficult to say why a design is poor. But it is extremely difficult to propose guidelines which will lead to the good design of a new system. For this reason, an important part of the project will be the evaluation of the guidelines.

The paper is organized as follows. In the next section two techniques are described, one primarily intended for design and the other primarily for implementation; these techniques together form a framework within which the system will be built. Section III contains a number of programming guidelines intended to help with the organization of the software; thus, these guidelines are primarily of use in the design phase, but will also apply to implementation when decisions have to be made. The final section examines the various stages of building systems, relates the stages to the programming guidelines and the organization of personnel, and establishes documentation procedures which will help to evaluate the project when it is over.

SECTION II

BASIC TECHNIQUES

As was stated previously, our fundamental concern is for producing reliable software systems. We will concentrate on two important aspects of the production problem. The first is the development of a system structure which copes with the inherent complexity of the system in an effective and understandable way; this will be done through the technique of modularity. Then, given the system structure, it is necessary to insure its clear and understandable representation in the system software; this will be accomplished through the technique of structured programming.

MODULARITY

To reiterate, a complex system is one in which there are so many system states that it is difficult to understand how to organize the program logic so that all states will be handled correctly. The obvious technique to apply when confronting this type of situation is "divide and rule." This is an old idea in programming and is known as modularization. Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas:⁽⁶⁾ "The connections between modules are the assumptions which the modules make about each other." Modules have connections in control via their entry and exit points; connections in data, explicitly via their arguments and values, and implicitly through data referenced by more than one module; and connections in the services which the modules provide for one another.

Traditionally, modularity was chosen as a technique for system production because it makes a large system more manageable. It permits efficient use of personnel, since programmers can implement and test different modules in parallel. Also, it permits a single function to be performed by a single module and implemented and tested just once, thus eliminating some duplication of effort and also standardizing the way such functions are performed.

The basic idea of modularity seems very good, but unfortunately it does not always work well in practice. The trouble is that the division of a system into modules may introduce additional complexity. The complexity comes from two sources: functional complexity and complexity in the connections between the modules. Examples of such complexity are:

- (1) A module is made to do too many (related but different) functions, until its logic is completely obscured by the tests to distinguish among the different functions (functional complexity).
- (2) A common function is not identified early enough, with the result that it is distributed among many different modules, thus obscuring the logic of each affected module (functional complexity).
- (3) Modules interact on common data in unexpected ways (complexity in connections).

The point is that if modularity is viewed only as an aid to management, then any ad hoc modularization of the system is acceptable. However, the success of modularity depends directly on how well modules are chosen. We will accept modularization as the way of organizing the programming of complex software systems. A major part of this paper will be concerned with the question of how good modularity can be achieved. First, however, it is necessary to define what good modularity is. This definition will be partially provided by defining modularity in terms of a new technique for organizing software: levels of abstraction.⁽³⁾

Levels of abstraction provide a conceptual framework for achieving a clear and logical design for a system. The entire system is conceived as a hierarchy of levels, the lowest levels being those closest to the machine. Each level supports an important abstraction; for example, one level might support segments (named virtual memories), while another (higher) level could support files which consist of several segments connected together. An example of a file system design model based entirely on a hierarchy of levels can be found in Madnick and Alsop.⁽⁷⁾

There are two important rules governing levels of abstraction. The first concerns resources (I/O devices, data): each level has resources which it owns exclusively and which other levels are not permitted to access. The second involves the hierarchy: lower levels are not aware of the existence of higher levels and therefore may not refer to them in any way. Higher levels may appeal to the functions of lower levels to perform tasks; they may also appeal to them to obtain information contained in the resources of the lower level.

Levels of abstraction differ from modules because a level consists of a group of related functions whereas a module is associated with only one function (at least externally). Levels of abstraction differ from modularity as previously defined because they are

accompanied by rules governing the connections between the levels. We will therefore change our definition of modularization as follows:

Modularization is defined to be the division of the system into a hierarchy of levels of abstraction, each level consisting of one or more externally accessible functions which share common resources. These levels are connected to one another in very simple and well-defined ways. Recall that connections exist both in control and in data. Connections in control are limited as follows:

- (1) Each function has only one entry point and always exits to the place from which it was invoked.
- (2) The rule about hierarchy of levels is observed.

Connections in data between two levels of abstraction are limited to the explicit arguments passed to a function and the values returned. Implicit interaction on common data may only occur among functions in the same level of abstraction.

STRUCTURED PROGRAMMING

Structured programming is a programming discipline which was introduced with reliability in mind.^(4,8) It is defined, and the rationale for it given, in Liskov and Towster;⁽¹⁾ only a brief summary is presented here. One justification for structured programming is that the resulting programs are easier to understand and to read than ordinary programs; this ease is then linked to the correctness of code before testing, thus increasing the chances of a reliable system.

There are three main rules which together define structured programming. The first defines the syntax of structured programs. A program may be thought of as made up of statements connected together by control structures. In structured programs only the following control structures are permitted: concatenation, selection of the next statement based on the testing of a condition, and iteration. Connection of two statements by a goto is not permitted. The statements themselves may be assignment statements or procedure calls.

The second rule, concerning how structured programs should be written, is the most important. It states that programs should be developed from the top down. The highest level of a program describes the flow of control among major functional components of the program;

names are introduced to represent these components. These names can be associated with code later; this code describes the flow of control among still lower level components, where again names are introduced to represent the components. The process stops when no undefined names remain. Each expansion of a name is called a module;¹ it is a goto-free program, having one entry point, and always exiting to the statement immediately following the one which refers to its name.

The third rule limits the size of structured program modules. The success of structured programming is based in large part on the readability of the resulting code. For this reason, modules are limited in size so that an entire module can be easily read and understood. A size of one computer printout page per module is suggested.⁽⁴⁾

Structured Programs and Higher Level Languages

A language which supports structured programming is a special kind of higher level language. From the very start it was obvious that we wanted to write the software in a higher level language, since readability and understandability of software were primary project goals. In addition, there is a general trend nowadays toward writing software, even for systems like operating systems, in a higher level language.⁽⁹⁾ The primary motivation is programmer productivity, which is considered more important than the efficiency of the code produced. The problem of inefficient code is in any case being alleviated by designing special languages for programming operating and similar systems.^(10,11) A language of this sort which also supports structured programming has been designed and implemented for this project.

Structured Programming and Proofs of Correctness

The following connection exists between structured programs and proofs of correctness. Before code is written for a module, a specification of the module exists which explains the input and output of the module and the function which it is supposed to perform. A form for this specification will be given in Section IV. When the module is coded, it is expressed in terms of specifications of lower level modules. The theorem to be proved is that the code of the module matches its specification; this proof will be given based on

¹Originally these expansions were called segments,⁽⁴⁾ but "segment" now means "named virtual memory," so it seemed advisable to select a different name. It is hoped that no confusion will result from the choice of "module," which is no longer being used in this paper to define modularity.

axioms stating that lower level modules match their specifications. The proof should not be too difficult because the module itself is small and logically straightforward due to the omission of goto's.

Structured Programs and Levels of Abstraction

A level of abstraction is made up of functions (or procedures), some of which may be referenced by other levels of abstraction (the external functions) while others (the internal functions) are used only within the level to perform certain tasks common to all work being performed by the level. Associated with each such function will be a structured program module, and the name of the function will be the same as the name of the module. In addition, however, modules are sometimes introduced in order to clarify the logic of a given function; thus, a function may be associated with more than one module.

Structured Programming and System Design

Structured programming is obviously applicable to system implementation. However, it is also a valuable aid for system design. Structured programs can replace flowcharts as a way of specifying what a program is supposed to do. It is no more difficult to write a structured program than a flowchart, since both contain approximately the same level of detail. The advantage of the structured program is that it is part of the final program; no translation is necessary (with the attendant possibility of introduction of errors). In addition, the structured program is more rigorous than a flowchart. For one thing, it is written in a programming language and therefore the semantics are well defined. For another, a flowchart only describes the flow of control among parts of a system, but a structured program defines the arguments and values of a referenced module. If a change in level of abstraction occurs at that point, then the connection between the two modules is completely defined by the structured program. This should help to avoid interface errors usually uncovered at system integration.

The way structured programs are written is very close to modularization as it is traditionally defined; in both cases the work to be performed is divided among lower level subprograms. This closeness is illustrated by calling these subprograms modules (this word is no longer being used for the definition of modularization). This means that structured programming is a particularly good environment in which to perform modularization. Structured programming does not explain how modules are to be grouped into levels of abstraction; this grouping occurs as a result of concepts about the system which are developed independently of the structured programs. But structured programs do provide a good way of expressing the system as a program as it develops.

SECTION III

PROGRAMMING GUIDELINES

In the preceding section, modularization was redefined in terms of levels of abstraction. Within this framework, the success of modularization depends on how well the levels of abstraction are selected. We will now present a tentative definition of good modularization supporting the goal of software reliability.

A good modularization satisfies the following requirements:

- (1) It satisfies the definition of modularization given in the preceding section and summarized here for convenience. The system is divided into a hierarchy of levels of abstraction, each level consisting of one or more externally accessible functions which share common resources. The connections in control among the levels are limited by the rule about hierarchy of levels. Connections in data are limited to the explicit arguments passed to the functions in the levels and the values returned.
- (2) The combined activity of the functions in a level of abstraction supports a single abstraction and nothing more. For example, a level of abstraction supporting files composed of many virtual memories should not contain any code supporting the existence of the virtual memories. The result of this restriction is that each level is substantially less complex than the system as a whole.
- (3) The system structure, which is defined by the way control passes among the functions of the levels of abstraction, is logically clear and understandable and is expressed by a structured program.

A system modularization satisfying the above requirements is compatible with the goal of software reliability. Since the system structure is expressed as a structured program, it should be possible to prove that it satisfies the system specifications, assuming that the structured programs which will eventually support the functions of the levels of abstraction satisfy their specifications. In addition, it is reasonable to expect that exhaustive testing of all relevant test cases will be possible. Exhaustive testing of the whole system means that each level must be exhaustively tested, and all combinations of levels must be exhaustively tested. Exhaustive testing of a single level involves both testing based on input parameters to the functions in the level

and testing based on intermediate values of state variables of the level. When this testing is complete, it is no longer necessary to worry about the state variables because of requirement 1. Thus, the testing of combinations of levels is limited to testing the input and output parameters of the functions in the levels. In addition, requirement 2 says that levels are logically independent of one another; this means that it is not necessary when combining levels to test combinations of the relevant test cases for each level. Thus, the number of relevant test cases for two levels equals the sum of the relevant test cases for each level, not the product.

GUIDELINES FOR SELECTING LEVELS OF ABSTRACTION

Now that we have a definition of good modularization, the next question is how can a system modularization satisfying this definition be achieved. The traditional technique for modularization is to analyze the execution time flow of the system and organize the system structure around each major sequential task. This technique leads to a structure which has very simple connections in control, but the connections in data tend to be complex (for examples see Parnas⁽¹²⁾ and Cohen⁽¹³⁾). The structure therefore violates requirement 1; it is likely to violate requirement 2 also since there is no reason (in general) to assume any correspondence between the sequential ordering of events and the independence of the events.

If the execution flow technique is discarded, however, we are left with almost nothing concrete to help us make decisions about how to organize the system structure. The guidelines presented here are intended to rectify this situation. These guidelines tend to overlap, and when designing a system, the choice of a particular level of abstraction will generally be based on several of the guidelines. Following the guidelines, an example of the selection of a particular level of abstraction within the Venus system⁽¹⁴⁾ is presented to illustrate the application of several of the principles; an understanding of Venus is not necessary for understanding the example.

Useful Abstractions

The most important reason for introducing a level of abstraction is as support for a useful abstraction. Abstraction is a very valuable aid to ordering complexity. Abstractions are introduced in order to make what the system is doing clearer and more understandable; an abstraction is a conceptual simplification because it expresses what is being done without specifying how it is done. Whenever a useful abstraction is identified, a level will be introduced to support it. Some abstractions express system features useful to the users

of the system, while others are primarily intended to aid the system designers, and the system users may never be aware of them. Examples of useful abstractions are: spooling of a shared device such as a card reader or printer, processes (see page 12), or virtual memories.

The following types of useful abstractions are to be expected when designing a system:

Abstractions of Resources

Every hardware resource available on the system will be represented by an abstraction having useful characteristics for the user. This abstraction will be supported by a level of abstraction whose functions map the characteristics of the abstract resource into the characteristics of the real underlying resource or resources. (This mapping may actually occur over several levels of abstractions.) For example, in an interactive system "abstract teletypes" with end of message and erasing conventions are to be expected.

Abstract Characteristics of Data

In most systems the users are interested in the structure of data rather than (or in addition to) storage of data. The system can satisfy this interest by the inclusion of a level of abstraction which supports the chosen data structure; functions of the level will map the structure into the way data is actually represented by the machine (again this may be accomplished by several levels). For example, in a file management system such a structure might be an indexed sequential access method.

Simplification of Levels of Abstraction

According to requirement 2, the functions comprising a level of abstraction support only one abstraction and nothing more. Sometimes it is difficult to see when this restriction is being violated. The two following guidelines are intended to help the system designer satisfy requirement 2:

Simplification Via Recognition of Common Functions

One candidate for a level of abstraction is a function (or group of functions) which is obviously going to be generally useful. Separating such groups is a common technique in system implementation and is also useful for error avoidance, minimization of work, and standardization. The existence of such a group simplifies other levels, which need only appeal to the functions of the lower level rather than perform the tasks themselves.

Sometimes a level of abstraction already exists which supports tasks very similar to some work which must be performed. When this is true, every effort should be made to use the functions of this other (lower) level, provided that this use does not force additional complexity on those functions.

Simplification Via Limiting Information

Another way of simplifying levels of abstraction is to limit the amount of information which they need to know (or even have access to). An example of such information is the formatting of data which is peripheral to the true function of the level (the data would be a resource of the level). The level may require the information embedded in the data but need not know how it is derived from the data (or possibly even where it is). This knowledge can be successfully hidden within a lower level of abstraction whose functions will provide requested information to higher levels when called; note that the data in question becomes a resource of the lower level.

System Maintenance and Modification

Producing a system which is easily modified and maintained is one of the primary goals of the project. This goal can be aided by separating into independent levels of abstraction functions which are performing a task whose definition is likely to change in the future. For example, if a level is paging data between core and some backup storage, it may be wise to isolate as an independent level of abstraction those functions which actually know what the backup storage device is. Then if a new device is added to the system (or a current device is removed), only the small independent level of abstraction need be changed; the larger level will already have been isolated from such changes by the requirement about connections between levels.

A Special Guideline for Asynchronous Systems

Not all complex systems need be asynchronous; however, systems which must deal with asynchronous events (for example, input/output interrupts) will necessarily be asynchronous. In addition, systems which need not be asynchronous are sometimes built to be asynchronous for reasons of efficiency (two or more events can then occur simultaneously). The file management system to be built for this project will be asynchronous. A special guideline is presented for simplifying the design of asynchronous systems.

A Model for Asynchronous Systems

Recent work in operating systems (3,14,15,16) has defined a model for asynchronous systems. This model both simplifies the conceptual difficulties of asynchronous systems and also reduces the amount of code required. It is based upon the concept of the work of the system being performed by a community of cooperating processes. The processes communicate and synchronize with one another by means of primitives provided by a very low level of abstraction which also serves to establish the existence of the processes; this level defines the system nucleus. P and V operations on semaphores are an example of such primitives; more useful communication devices, such as queues or mailboxes, can be provided at a slightly higher level.

The advantage of the communicating process approach is that it allows the many system tasks which are logically asynchronous to be handled in a physically asynchronous manner. This leads to clarity and reduced complexity in the design, which in turn reduces the complexity required of the implementation. Since complexity is a primary obstacle to building correct systems, reduction of complexity cannot help but aid the goal of system reliability.

Cooperating processes are related to levels of abstraction in the following way. Whenever control passes from one level of abstraction to another, this may occur either by calling a procedure or by synchronizing with another process. Within a level, however, probably only calls are legal, for the reason that the decision to change processes is a major one and probably coincides with a change in level.

The Guideline

The previous model should be used when designing the file management system.

An Example from Venus

The following example from Venus is presented because it illustrates many of the points made about selection, implementation, and use of levels of abstraction. The concept to be discussed is that of external segment name, referred to as ESN from now on.

The concept of ESN was introduced as an abstraction primarily for the benefit of users of the system. The important point is that a segment (named virtual memory) exists both conceptually (as a place where a programmer thinks of information as being stored) and in reality (the encoding of that information in the computer). The reality of a segment is supported by an internal segment name (ISN)

which is not very convenient for a programmer to use or remember. Therefore, the symbolic ESN was introduced.

As soon as the concept of ESN was imagined, the existence of a level of abstraction supporting this concept was implied. This level owned a nebulous data resource, a dictionary, which contained information about the mappings between ESNs and ISNs. The formatting of this data was hidden information as far as the rest of the system was concerned. In fact, decisions about the dictionary format and about the algorithms used to search a dictionary could safely be delayed until much later in the design process. A collective name, the dictionary functions, was given to the functions making up this level of abstraction.

As soon as the ESN level existed, it was necessary to define the interface presented by this level to the rest of the system (using a structured programming technique for design, a level would probably begin to exist when the first function within this level was specified). Obvious items of interest are ESNs and ISNs; the format of ISNs was already determined, but it was necessary to decide about the format of ESN. The most general format would be a count of the number of characters in the ESN followed by the ESN itself; for efficiency, however, a fixed format of six characters was selected.

At this point a generalization of the concept of ESN occurred, because it was recognized that a two-part ESN would be more useful than a single symbolic ESN. The first part of the ESN is the symbolic name of the dictionary which should be used to make the mapping; the second part is the symbolic name to be looked up in the dictionary. This concept was supported by the existence of a dictionary containing the names of all dictionaries. A format had to be chosen for telling dictionary functions which dictionary to use; for reasons of efficiency, the ISN of the dictionary was chosen (thus avoiding repeated conversion of dictionary ESN into dictionary ISN).

At this point we had the identification of a level of abstraction. We knew what type of function belonged in this level, what sort of interface it presented to the rest of the system, and what information was kept in dictionaries. As the system design proceeded, new dictionary functions were specified as needed. Two generalizations were realized later. The first was to add extra information to the dictionary; this was information which the system wanted on a segment basis, and the dictionaries were a handy place to store it. The second was to make use of dictionary functions as a general mapping device; for example, dictionaries are used to hold information about the mapping of record names into tape locations, permitting simplification of the higher level.

In reality, as soon as dictionaries and dictionary functions were conceived, a core of dictionary functions were implemented and tested. This is a common situation in building systems and did not cause any difficulty in this case. For one thing, extra space was purposefully left in dictionary entries because we suspected we might want extra information there later although we did not then know what it was. The algorithm selected was straight serial search; the search was embedded in two internal dictionary functions (not available for calling from outside the level) so that the format of the dictionary may be changed and the search redefined with very little effect on the system or most of the dictionary functions. This follows the guideline of modifiability.

SECTION IV

GUIDELINES FOR BUILDING SYSTEMS

This section describes the guidelines to be followed when building the file management system. The programming guidelines of Section III will be applied whenever decisions are being made. This section is concerned with giving motivations for decisions within the various stages of building a system. In addition, management guidelines are given governing the way that personnel will be used in each stage and also specifying administrative techniques for keeping track of the development of the system and the application, success or failure of the various guidelines.

As was mentioned in the introduction, it is customary to divide the building of a system into three phases: design, implementation and testing. In this section we will distinguish two phases: a design phase, and an implementation and testing phase. This division is based on the organization of personnel: one organization is required for the design phase, and another is used for both implementation and testing. Personnel are organized into a structure which will hopefully permit global design considerations to control local decisions and which will encourage informal proofs of correctness, thus enhancing the chances for a reliable system.

The design phase consists of making global decisions which affect the system as a whole and representing these decisions in structured programs. Thus design is concerned with identification of levels of abstraction and the connections between them. Implementation includes making local decisions within a level of abstraction and representing them by structured programs; these decisions may even involve the introduction of new levels, which will, however, be completely hidden from the rest of the system by the level being implemented. Thus implementation includes making design decisions; however, fewer global considerations will be required, which means the decisions are not as difficult to make.

In the course of building a system, a point is reached when the design is considered to be complete; an attempt will be made to capture the characteristics of the system at that time. This distinction is important for determining what constraints to put on subcontractors and should also be useful for analyzing the progress of software being produced in-house.

THE DESIGN PHASE

The design phase is the most difficult part of building a system because it is at this point that the full inherent complexity of the system is encountered. It is also the most critical phase, because all important (i.e., global) decisions are made at this time and a general outline of the way in which the implementation should proceed is given. If the design is bad, then the problems encountered within implementation and testing are increased and hopes of obtaining a reliable system diminish. It is during the design phase that the programming guidelines of Section III will be most relevant.

So it is important in the design phase to achieve a "good" system design, satisfying the following working definition:

- (1) The system will satisfy its requirements.
- (2) The design produces a good system modularization as defined in Section III.

Obviously this is a very vague definition; if this project can produce a better one, that will be a substantial contribution.

When system design begins, descriptions of the services which the system is supposed to provide are given. An important part of system design is to turn these descriptions into precise specifications which will support the building of the system. Care must be taken that none of the original intent of what the system was supposed to do is lost in this process.

In addition to defining initial system specifications in light of expected user services (these are often referred to as top-level considerations), the initial design phase will also be concerned with the hardware on which the system is going to run (bottom-level considerations). The concern with hardware may include the selection and purchase of the hardware; in any case a thorough understanding of the characteristics of the hardware will be necessary so that the evolving design will be consistent with these characteristics. Thus the purpose of the design phase is to produce a system design satisfying system specifications in a way which is compatible with the hardware on which the system is going to run. The design will be exhibited as a system modularization as defined in Section II; the guidelines of Section III will be used to help identify what this modularization should be.

How to Proceed with the Design

The very first phase of the design (phase 1) will be concerned with defining precise system specifications and analyzing the hardware requirements of these specifications. The result of this phase will be a number of abstractions which represent the eventual system behavior in a very general way. These abstractions imply the existence of levels of abstractions, but very little is known about the connections between the levels, the flow of control among the levels, or how the work of the levels will be accomplished. Examples of such abstractions from Venus are: spooling of the card reader and printer; limited ownership of teletypes; existence of an executive to control jobs; existence of a loader to run jobs; etc. Every important external characteristic of the system should be present as an abstraction at this stage. Many of the abstractions have to do with the management of system resources; others have to do with services provided to the user. One abstraction to be expected in an interactive system is that of a general command language; when this is missing, the result is likely to be a confusing duplication of commands meaning different things in different environments.

The second (and final) phase of system design (phase 2) investigates the practicality of the abstractions proposed by phase 1 and establishes the data connections between the levels and the flow of control among the levels. This latter exercise establishes the placement of the various levels in the hierarchy. The second phase occurs concurrently with the first; as abstractions are proposed, their utility and practicality are immediately investigated. A level has been adequately investigated when its connections with the rest of the system are known and when the designers are confident that implementation of the abstraction is practical. Varying depths of analysis are necessary; for example, the ESN level of abstraction requires almost no analysis, while spooling requires much analysis with the resulting identification of several internal levels of abstraction.

Design of an Asynchronous System

The design of an asynchronous system should be based on the model described in Section III. This means that the functions performed by the system nucleus as well as the meaning of such concepts as "process" should be identified early in the design phase since they represent a basis upon which the whole system design will rest. At this point, it is necessary to define the primitives to be used by system processes to synchronize with each other and to share system resources (these primitives need not be made available to the users of the system). These primitives may be supported directly by the system nucleus, or they may be supported by a very low level of abstraction using primitives of the nucleus.

Special Problems Which Must Be Solved During the Design

A few difficult problems which must be solved during the design phase are enumerated here. It is not clear that solutions to some of these problems fit into levels of abstraction or are handled well by structured programs.

- 1) System startup and shutdown. One problem of system startup is the initialization of the resources of the various levels of abstraction. This is best solved by including in each level of abstraction a special function whose job is to provide initialization for that level. A similar function may be required to shut down the level.
- 2) Error Protection. The system must be protected from the effects of user errors. A partial solution to this problem is to localize the effect of errors as much as possible; this is also helpful to the user for debugging his program. System errors should also be localized if possible, and error handling by the levels of abstraction is an important part of the design of the system.
- 3) Error Recovery. This is a very difficult problem. Solutions are based on redundancy of information and protection of critical data by frequent snapshots. Information must be kept about ownership of resources if "warm" starts are desired.
- 4) Efficiency. There are many types of efficiency, and there are often tradeoffs between the different types. We are considering efficiency of performance here. A system often has performance criteria to meet, and it is necessary to evaluate a given system structure in depth to be sure it will support these criteria. In general, however, system efficiency is very difficult to define during the design phase (especially in an asynchronous system) because the tradeoffs are not yet clear, and the designers should beware of spending too much time in optimizing any one part of the system. A better approach is to make the system as modifiable as possible, intending to tune it after it is built.
- 5) Instrumentation. It is only possible to tune the system if its behavior can be measured. As the system design evolves, some attention should be paid to the question of what types of measurements are required. Most important is the requirement for the existence of a very low level of abstraction (possibly supported by hardware) which permits measurement of the behavior of the system without changing that behavior significantly or at all.

- 6) Debugging Aids. An interactive system must provide interactive debugging aids for its users. It is not clear where such aids fit in in the hierarchy of levels.
- 7) Sharing of resources among the users. The system must be designed to insure that the available resources (including data) are shared among the users in a deadlock-free manner.
- 8) Sharing of data resources within the system. The system may share data differently from the users for reasons of efficiency. Again care must be taken to avoid deadlock; ownership of only one resource at a time or some ordering on the way ownership can be obtained are both satisfactory solutions.
- 9) Synchronization. The synchronization of the processes making up the system must occur in the correct order.

Structured Programming

It is not clear exactly how early structured programming of the system should begin. Obviously, whenever the urge is felt to draw a flowchart, a structured program should be written instead. Structured programs connecting all the levels of abstraction together will be expected by the end of the design phase. The best rule is probably to keep trying to write structured programs; failure will indicate that system abstractions are not yet sufficiently understood and perhaps the effort will shed some light on where more effort is needed.

Specifications of Levels and Functions

As the structured programs are written, names are introduced to represent lower level modules. Often these names will stand for functions of lower levels of abstraction. In this way levels of abstraction come into existence and are linked to existing levels. As the names are introduced, specifications should be given describing what the proposed module should do.

The concept of system reliability as discussed in this paper is based on informal proofs of correctness that code matches specifications. It is important, therefore, that specifications be complete and understandable. We discuss here the information to be contained in such specifications. Specifications will be given for both levels of abstraction and for the functions within the levels; in this way the grouping of functions into levels is emphasized. Not all the information discussed below will be known in the design phase; much information will be added during implementation (including the addition of many new functions and possibly even new levels of abstraction).

The following information should be contained in the specification of a level of abstraction:

- 1) the name of the level;
- 2) a description of the abstraction which it supports;
- 3) a list of the hardware resources owned by the level, if any;
- 4) a list of the data resources owned by the level (this includes the data holding information about the state of the level);
- 5) information about the placement of the level within the hierarchy of levels and whether the level occupies its own process;
- 6) a list of the functions of the level which are externally accessible;
- 7) a list of the functions of the level which are internally accessible (probably not known until implementation).

The following information should be contained in the specification of a function:

- 1) the name of the function;
- 2) the name of the level of abstraction to which the function belongs and whether the function is external or internal;
- 3) a description of every argument passed to the function and every value returned. In each case, the legal bounds on the arguments and values should be carefully delineated.
- 4) a description of what the function does (not how it works) based on its arguments and values, including the handling of errors;
- 5) the expectations of the function about the state of the resources of the level when it is entered and its effect on that state, including error handling (probably not known until implementation).

End of the Design Phase

The design will be considered finished when the following criteria are satisfied:

1. All major levels of abstraction have been identified, the system resources have been distributed among them, and their positions in the hierarchy established. It is known how the levels are distributed among the processes.
2. The system exists as a structured program, showing how the flow of control passes among the levels of abstraction. The structured program consists of several modules, but no module is likely to be completely defined (in the sense of being ready to execute). Specifications exist for all levels and functions. The interfaces between the levels have been defined, and the relevant test cases for each level have been identified.
3. Sufficient information is available so that a skeleton of a user's guide to the system could be written. Many details of the guide would be filled in later, but new sections should not be needed.

Organization of Personnel

The design phase is accomplished by the design team, which contains everyone connected with the design in some capacity. Within this team is a very small design team core which has the responsibility for producing the design. Members of the core must have a global knowledge of the system and will use this knowledge first to generate the abstractions of phase 1 of the design and then to decide between alternative solutions produced through phase 2 analysis. The core should contain at least two members, so that global system considerations can be discussed intelligently and thus provide a check on the correctness of the developing design. However, the core should be small to avoid wasted time and energy. One member of the core will be designated the project leader (in reality, the responsibilities of the leader may be distributed among members of the core).

The team core is first of all responsible for suggesting the abstractions of phase 1 of the design, although suggestions of members of the design team will certainly be welcome. The core will also decide what type of phase 2 analysis these abstractions require and will direct that this analysis be carried out by selected members of the design team (including core members). Examples of types of analysis range from asking a team member or members to develop a solution to support an abstraction in order to determine whether implementation is practical, to asking them to examine several alternative methods of accomplishing an abstraction and then to present the strengths and weaknesses of the alternatives. Finally, the core is responsible for writing the structured program or programs which tie the levels of abstraction together.

Each member of the design team is responsible for performing phase 2 analysis as directed by the core. At the beginning of the design he may have as much knowledge about the global requirements of the system as members of the core, but as the design progresses, this will probably not be true as more and more of his time is spent in performing detailed analysis of a part of the design. Therefore, he must expect to present a justification for his decisions to the design core, who will evaluate them with respect to global system considerations. Sometimes his decisions will conflict with these considerations (although the core should explain relevant considerations to him in advance to avoid this as much as possible). In this case his work will have to be redone. If his analysis has gone very deep, considerable effort will be wasted. For this reason, the core will try to specify a depth of analysis in advance, and the team member must resist the temptation to exceed this depth. A depth of analysis such that design decisions are being made based on earlier decisions which have not been approved is probably too deep.

Design Meetings and Project Documentation

There will be frequent meetings during the design phase. These meetings will always be attended by members of the design team core; some members of the design team may also attend. In the early stages of the design these meetings are likely to be very informal as abstractions are proposed, decisions about hardware are made, certain abstractions are selected for investigation, and representation of the system as a structured program is attempted. Later in the design the meetings should become more orderly (as the design itself becomes better defined); meetings will consist of presentations of phase 2 analysis followed by design decisions or requests for more analysis.

In addition to team members, meetings will always be attended by a secretary. An important responsibility of the project leader is to keep a history of the project in a design notebook; one facet of that history will be a record of all design meetings. The secretary's job will be to help as much as possible with the preparation of the notebook. The notebook should contain a brief but lucid history of the project; for this reason, verbatim meeting notes may not be desirable. Instead a brief description of a problem under discussion should be dictated to the secretary; such dictations will occur throughout the meetings. The notebook should include information about the basis for every design decision, together with alternative solutions rejected and why. In addition to records of design meetings, the design notebook should contain a record of any other significant development; for example, descriptions of spontaneous discussions and whatever else the project leader considers significant.

It is very likely that as the design continues inadequacies in the guidelines will be detected. Notes of these inadequacies, together with any changes which seem reasonable, should be entered in the guide-line notebook. The team leader has the responsibility not only to make sure that these entries are made but also to examine current procedures in order to detect inadequacies.

IMPLEMENTATION AND TESTING

Our guidelines for implementation and testing will be based to a large extent on the work of Harlan Mills.⁽⁴⁾ These techniques have already been partially verified by work performed at MITRE in the building of the SPIL compiler.

Mills's techniques for implementation and testing were defined with reliability in mind. Implementation is carried out from the top down, using structured programs. Thus, modules are specified and the interfaces between modules defined before the code supporting the modules is written. In addition, the modules are structured in such a way that the flow of control is emphasized, and modules are limited in size to less than one page. The result is that the programs are very readable. This readability is then exploited by requiring that modules be read by someone other than the programmer; this technique is bound to uncover many errors.

Testing does not begin until an entire subsystem has been coded. The subsystem is tested within the framework of the entire system. This means that it is not necessary to write testing programs which drive the various modules; on the other hand, it may be necessary to write program stubs to stand for parts of the system which are not yet defined. One advantage of this approach is that the central logic of the system is tested the most, since it is run every time a new subsystem is tested. The primary advantage, however, is that this technique eliminates system integration problems almost entirely. When a newly coded subsystem is tested, it must be integrated with all the parts of the system which are already defined; thus, each part is integrated only once.

We will modify Mills's techniques in a few significant ways. In the first place, we will be working within a hierarchy of levels of abstraction. Thus, we will talk about implementation of levels rather than of subsystems. However, testing of levels will occur (as much as possible) within the framework of the entire system as defined by Mills.

In addition, we will not limit ourselves to a strict top-down implementation and testing, because this is not practical in the production of large systems, which must generally be completed within a certain time period. In order to satisfy time constraints, it is often necessary to implement parts of the system in parallel, and we intend to attempt this in order to see if it is compatible with reliability. Even without parallel implementation, it is not clear that top-down implementation is always the best; for example, it might benefit production if certain low levels of abstraction, which are well-defined and widely used, were implemented and tested first. Therefore, we will make decisions about the order of implementation and testing based on tradeoffs between practicality and reliability.

Design Within Implementation

It must be recognized that many design decisions will still remain when implementation begins. Also the work that the level performs may still be very complex, requiring further modularization. The person performing the design should rely on the programming guidelines to help him make decisions. He must understand where the level fits in the hierarchy and what services are performed for it by lower levels. Care must be taken that the resulting design makes use of these services and that it does not violate the global considerations of the system (management guidelines will be set up to insure this).

Order of Implementation

Implementation will be carried out on an entire level of abstraction at a time. Ordinarily implementation will not begin until the design phase is over; however, for certain particularly well-defined levels, it may be practical to begin implementation before this (as, for example, with the dictionary functions in Venus). The danger in early implementation is that changes in the functional specifications may be made as the design progresses, thus invalidating the implementation, so the decision to implement early should be made very cautiously. Within a level, implementation should be entirely top-down, following the rules governing structured programming.

When the design is complete, an order of implementation of the levels must be selected. Generally, higher levels should be implemented first. For one thing, certain design decisions may still remain which will have to be made as the implementation progresses; it is better if these decisions are made during implementation of high levels of abstraction because then it is easier to accommodate global considerations. The exceptions to this order are low levels which are essentially independent of most of the design and which provide widespread support for higher levels. It does not matter when such levels are implemented, but they should probably be tested very early in order to minimize the testing effort.

The order of implementation suggested here is neither top-down nor bottom-up, but rather a combination of both. Implementation should be bottom-up when it is obvious that this is the right way to proceed; however, when there is a choice, top-down (highest levels first) implementation should be the rule. This is different from the usual order of implementation which is generally always bottom-up. Simultaneous implementation is possible for levels which are independent of one another (the design decisions required do not depend on one another).

Specifications

New information will become available as implementation progresses. This information should be added to the specifications which were started during the design phase (see previous section); specifications should also be given for all new levels and functions defined during implementation.

Order of Testing

Testing will occur on entire levels of abstraction at a time and will not begin until the entire level has been implemented. The order of testing need not be the same as the order of implementation, but generally the same order will be a good idea. This means that with the exception of certain low levels, testing will occur on higher levels before lower levels. Top-down testing will obtain the advantages discussed earlier in this section. However, two questions immediately arise: 1) It is most important that all relevant test cases be tested; will this occur naturally in top-down testing? 2) Top-down testing implies simulation of lower levels (rather than the standard simulation of higher levels); how much effort does this require? We hope to answer these questions while building the file management system.

Organization of Personnel

A hierarchical structuring of personnel seems reasonable for implementation and testing. At the top is the project leader whose responsibility is to resolve conflicts between implementations of levels and global considerations of the system as a whole. The project leader also decides when and in what order levels of abstraction should be implemented and tested. The actual implementation and testing of a level of abstraction (or possibly several connected levels) is carried out by an implementation and testing team. One member of the team is the team leader; this person is responsible for:

- 1) performing any design which may still be necessary. This design must be cleared with the project leader who will evaluate it with respect to global requirements. The design will be expressed in structured programs.
- 2) assigning pieces of the level to team members for implementation. He should not do any further implementation himself.
- 3) reading the structured programs produced by team members and satisfying himself that they are correct. He may ask team members to justify their code to help in this task.
- 4) designing tests for the level of abstraction. These tests must cover every relevant test case, as determined by the legal ranges of input and output values of procedures in the level. In addition, internal logic must be tested, and the team leader may request the implementer to help him define these tests.

Team members will implement procedures of the level as directed by the team leader. As soon as a procedure has been implemented, it becomes public and is entered in a notebook accessible to everyone. It may be compiled and corrected, with the changes becoming public as they occur, but no testing may be performed. If an implementer has questions, he may ask the team leader, or he may consult the public listings of other implemented procedures.

Since we intend to implement some of the levels in parallel, more than one implementation and testing team will be active at once. It is likely that project members will serve on more than one team, and they may do so in different capacities. In particular, the leader of one team could be an ordinary member of another team.

The Programming Secretary

In a system of any size there is a lot of work involved in simply getting new information to those who need it, keeping decks up to date, and so forth. The requirement that code be public adds to this burden. If programmers must perform these tasks, a considerable portion of their time is used in non-productive ways. In addition, there is the danger that the work will not be done, and this work is essential to the reliability of the system. Therefore, the project should have the services of a clerical person to do this work; we will call this person the programming secretary as Mills⁽⁴⁾ does. The programming secretary will perform all clerical tasks associated with the project; this includes use of the computer to edit and compile programs, but ability to program is not necessary. Secretarial skills

are much more desirable. The programming secretary should work directly for the project leader and could also do the secretarial tasks discussed in the design phase; the work to be performed (in order of priority) is:

- (1) attend design meetings and update the design notebook and other project documentation as directed by the project leader;
- (2) perform programming secretary tasks; and
- (3) do other secretarial work on a low priority basis.

Project Documentation

The design notebook and guideline notebook started in the design stage should be continued in the implementation and testing stage. In addition, two new notebooks will be made; one containing a Test History, and one containing an Error History.

Design Notebook

This notebook contains a chronological history of the development of the system. During the implementation and testing stage, the following information should be included in the design notebook:

- (1) notes of all meetings. Generally brief summaries of decisions rather than verbatim notes will be included. Meetings will become less frequent as this stage progresses, but will still be called when design decisions need to be made. One reason for such decisions will be an analysis of design-within-implementation decisions with respect to global considerations. Another, which will hopefully not occur, will be design errors discovered as implementation and testing proceeds.
- (2) Chronological information such as:
 - (a) date implementation of a level began (including analysis of why this particular order of implementation was chosen);
 - (b) size of implementation and testing team;
 - (c) date implementation was finished;
 - (d) date testing began (and analysis of order);

- (e) date testing finished. At this point some indication of the difficulty of generating the test environment should be included.

Guideline Notebook

This notebook should be continued as in the design phase. An analysis of the success or failure of the guidelines will be an important part of the entries in the guideline notebook. Information about the success or failure of the management policies should be included.

Test History

This notebook will include information about the running of tests for the various levels. Its purpose is to make this information available so that if the system is modified in the future, tests for unaffected levels can be rerun as insurance that the modifications did not harm them in any way. Entries will include: an analysis of relevant test cases, input testing these cases (decks will also be saved), and a history of successful runs on these cases, with dumps of significant information.

Error History

Many of the management procedures described in this paper are aiming at the elimination of errors before testing begins. Therefore, errors found during testing will be an indication of failure. Such discoveries may be very serious, possibly indicating design errors; information about these errors (if any) will be entered in the design notebook. All other errors are implementation errors (although such errors may also be serious in the sense that much recoding will be required).

An entry should be made in the Error History notebook for each implementation error uncovered during testing. This entry should tell:

- (1) name of level of abstraction being tested;
- (2) name of procedure in which error occurred;
- (3) an analysis of the error. This analysis should be fairly specific; i.e., "logic error" is not enough information.
- (4) it is possible that the error uncovered occurs in a different level of abstraction than the one being tested. This other level will already have been tested and this would indicate

that some relevant test case had been ignored in those tests. If this occurs, an analysis of the failure of the test should be included (of course, a new test should be added to the test history notebook as a result).

Documentation of the System

Two forms of documentation of a system are customary: user documentation and system documentation. The user documentation describes the services provided by the system and tells the user how to make use of these services; it explains very little about how the services are provided. An outline of this document should be available from the design phase; sections will be filled in as implementation progresses. New sections should not be needed, since these would indicate some aspect of the system not considered in the design. If a new section is needed, an analysis of why it is needed should be entered in the design notebook.

System documentation is intended for the systems programmer who is going to modify or maintain the system. It must contain sufficient information to permit him to find the part of the system which concerns him, to understand that part within the framework of the system as a whole, and to understand the logic of the part itself. We are hoping that the structured programs together with the function and level specifications will constitute an adequate system documentation. When the system is complete, we will analyze the structured programs and specifications to see if they are in fact adequate, add any information which is lacking, and make an entry in the guideline notebook describing the additional information required.

REFERENCES

1. B. H. Liskov and E. Towster, "The Proof of Correctness Approach to Reliable Systems," The MITRE Corporation, ESD-TR-71-222, Bedford, Massachusetts, July 1971.
2. E. W. Dijkstra, "Structured Programming," Software Engineering Techniques, J. N. Buxton and B. Randell, (eds.), October 1969, 84-88.
3. E. W. Dijkstra, "The Structure of the "THE" - Multiprogramming System," Communications of the ACM, 11, 5, May 1968, 341-346.
4. H. D. Mills, "Structured Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin (ed.), Prentice Hall, Inc., Englewood Cliffs, New Jersey, 41-55.
5. M. Conway, "How Do Committees Invent?," Datamation, 14, 4, April, 1968, 28-31.
6. D. L. Parnas, "Information Distribution Aspects of Design Methodology," Technical Report, Department of Computer Science, Carnegie-Mellon University, February 1971.
7. S. Madnick and J. Walsop, II, "A Modular Approach to File System Design," AFIPS Conference Proceedings, 34, (1969), AFIPS Press, Montvale, New Jersey, 1-13.
8. E. W. Dijkstra, "Notes on Structured Programming," Technische Hogeschool, Eindhoven, The Netherlands, August 1969.
9. F. J. Corbato, "PL/1 as a Tool for Systems Programming," Datamation, 15, 5, (May 1969), 68-76.
10. W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," Communications of the ACM, 14, 12, (December 1971), 780-790.
11. B. L. Clark and J. J. Horning, "The System Language for Project SUE," Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation, SIGPLAN Notices, 6, 9, (October 1971), 79-85.
12. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Technical Report CMU-CS-71-101, Carnegie-Mellon University, August 1971.

13. A. Cohen, "Modular Programs: Defining the Module," Datamation, 18, 1, (January 1972), 34-37.
14. B. H. Liskov, "The Design of the Venus Operating System," Communications of the ACM, 15, 3, (March 1972), 144-149.
15. P. Brinch Hansen, "The Nucleus of a Multiprogramming System," Communications of the ACM, 13, 4, (April 1970), 238-250.
16. P. J. Denning, "Third Generation Computer Systems," Computer Surveys, 3, 4, (December 1971), 175-216.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) The MITRE Corporation P. O. Box 208 Bedford, Massachusetts 01730		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP	
3. REPORT TITLE GUIDELINES FOR THE DESIGN AND IMPLEMENTATION OF RELIABLE SOFTWARE SYSTEMS			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates)			
5. AUTHOR(S) (First name, middle initial, last name) B. H. Liskov			
6. REPORT DATE FEBRUARY 1973	7a. TOTAL NO. OF PAGES 36	7b. NO. OF REFS 16	
8a. CONTRACT OR GRANT NO. F19628-71-C-0002	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-72-164		
b. PROJECT NO. 671A	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) MTR-2345		
c.			
d.			
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Electronic Systems Division, AFSC L. G. Hanscom Field, Bedford, Mass. 01730	
13. ABSTRACT This document describes experimental guidelines governing the production of reliable software systems. Both programming and management guidelines are proposed. The programming guidelines are intended to enable programmers to cope with a complex system effectively. The management guidelines describe an organization of personnel intended to enhance the effect of the programming guidelines.			

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
PROGRAMMING TECHNIQUES						
SOFTWARE DEVELOPMENT						
SOFTWARE RELIABILITY						
STRUCTURED PROGRAMMING						