DESIGNING SOFTWARE FOR EASE OF EXTENSION AND CONTRACTION

David L. Parnas

Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

Designing software to be extensible and easily contracted is discussed as a special case of design for change. A number of ways that extension and contraction problems manifest themselves in current software are explained. Four steps in the design of software that is more flexible are then discussed. The most critical step is the design of a software structure called the "uses" relation. Some criteria for design decisions are given and illustrated using a small example. It is shown that the identification of minimal subsets and minimal extensions can lead to software that can be tailored to the needs of a broad variety of users.

I. INTRODUCTION.

This paper is being written because the following complaints about software systems are so common:

(A) "We were behind schedule and wanted to deliver an early release with only <proper subset of intended capabilities>, but found that that subset would not work until everything worked."

(B) "We wanted to add <simple capability>, but to do so would have meant rewriting all or most of the current code."

(C) "We wanted to simplify and speed up the system by removing the <unneeded capability>, but to take advantage of this simplification we would have had to rewrite major sections of the code."

(D) "Our SYSGEN was intended to allow us to tailor a system to our customers' needs but it was not flexible enough to suit us."

After studying a number of such systems, I have identified some simple concepts that can help programmers to design software so that subsets and extension are more easily obtained. These concepts are simple if you think about software in the way suggested by this paper. Programmers do not commonly do so.

II. SOFTWARE AS A FAMILY OF PROGRAMS.

When we were first taught how to program, we were given a specific problem and told to write one program to do that job. Later we compared our program to others, considering such issues as space and time utilization, but still assuming that we were producing a single product. Even the most recent literature on programming methodology is written on that basis. Dijkstra's "Discipline of Programming" [1] uses predicate transformers to specify the task to be performed by the program to be written. The use of the definite article implies that there is a unique problem to be solved and but one program to write.

Today, the software designer should be aware that he is not designing a single program but a family of programs. As discussed in an earlier paper [2], we consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them. This rather pragmatic definition does not tell us what pays, but it does explain the motivation for designing program families. We want to exploit the commonalities, share code, and reduce maintenance costs.

Some of the ways that the members of a program family may differ are listed below:

(1) They may run on different hardware configurations.

(2) They may perform the same functions but differ in the format of the input and output data.

(3) They may differ in certain data structures or algorithms because of differences in the available resources.

(4) They may differ in some data structures or algorithms because of differences in the size of the input data sets or the relative frequency of certain events.

(5) Some users may require only a subset of the services or features that other users need. These "less demanding" users may demand that they not be forced to pay for the resources consumed by the unneeded features.

Engineers are taught that they must try to anticipate the changes that may be made, and are shown how to achieve designs that can easily be altered when these anticipated changes occur. For example, an electrical engineer will be advised that the whole world has not standardized on 60-cycle, 110-volt current. Television designers are fully aware of the differing transmission conventions that exist in the world. It is standard practice to design products that are easily changed in those aspects. Unfortunately, there is no magic technique for handling unanticipated changes. The makers of conventional watches have no difficulty altering a watch that shows the day so that it displays "MER" instead of "WED," but I would expect a long delay for redesign were the world to switch to a ten day week.

Software engineers have not been trained in this way. The usual programming courses neither mention the need to anticipate changes nor do they offer techniques for designing programs in which changes are easy. Because programs are abstract mathematical objects, the software engineers' techniques for responding to anticipated changes are more subtle and more difficult to grasp than the techniques used by designers of physical objects. Further, we have been led astray by the other designers of abstract objects - mathematicians who state and prove theorems. When a mathematician becomes aware of the need for a set of closely related theorems, he responds by proving a more general theorem. For mathematicians, a more general result is always superior to a more specialized product. The engineering analogy to the mathematician's approach would be to design television sets containing variable transformers and tuners that are capable of detecting several types of signals. Except for U.S. armed forces stationed overseas, there is little market for such a product. Few of us consider relocations so likely that we are willing to pay to have the generality present in the product. My guess is that the market for calendar watches for a variable length week is even smaller than the market for the television sets just described.

In [2] I have treated the subject of the design of program families rather generally and in terms of text in a programming language. In this paper I focus on the fifth situation described above; families of programs in which some members are subsets of other family members or several family members share a common subset. I discuss an earlier stage of design, the stage when one identifies the major components of the system and defines relations between those components. We focus on this early stage because the problems described in the introduction result from failure to carefully consider early design decisions.

III. HOW DOES THE LACK OF SUBSETS AND EXTENSIONS MANIFEST ITSELF?

Although we often speak of programs that are "not subsetable" or "not extensible," we must recognize that phrase as inaccurate. It is always possible to remove code from a program and have a runable result. Any software system can be extended (TSO proves that). The problem is that these subsets and extensions are not the programs that we would have designed if we had set out to design just that product. Further, the amount of work needed to obtain the product seems all out of proportion to the nature of the change. The problems encountered in trying to extend or shrink systems fall into four classes.

A. Excessive information distribution.

A system may be hard to extend or contract if too many programs were written assuming that a given feature is present or not present. This can be illustrated by an operating system in which an early design decision was that the system would support three conversational languages. There were many sections of the system where knowledge of this decision was used. For example, error message tables had room for three entries. An extension to allow four languages would have required that a great deal of code be rewritten. More surprisingly, it would have been difficult to reduce the system to one that efficiently supported only two of the languages. One could remove the third language, but to regain the table space, one would have had to rewrite the same sections of code that would be rewritten to add a language.

B. A chain of data transforming components.

Many programs are structured as a chain of components, each receiving data from the previous component, processing it (and changing the format), before sending the data to the next program in the chain. If one component in this chain is not needed, that code is often hard to remove because the output of its predecessor is not compatible with the input requirements of its successor. A program that does nothing but change the format must be substituted. One illustration would be a payroll program that assumed unsorted input. One of the components of the system accepts the unsorted input and produces output that is sorted by some key. If the firm adopts an office procedure that results in sorted input, this phase of the processing is unnecessary. To eliminate that program, one may have to add a program that transfers data from a file in the input format to a file in the format appropriate for the next phase. It may be almost as efficient to allow the original SORT component to sort the sorted input.

C. Components that perform more than one function.

Another common error is to combine two simple functions into one component because the functions seem too simple to separate. For example, one might be tempted to combine synchronization with message sending and acknowledgment in building an operating system. The two functions seem closely related; one might expect that for the sake of reliability one should insist on a "handshake" with each exchange of sychronization signals. If one later encounters an application in which synchronization is needed very frequently, one may find that there is no simple way to strip the message sending out of the synchronization routines. Another example is the inclusion of run-time type-checking in the basic subroutine call mechanism. In applications where compile-time checking or verification eliminates the need for the run-time type-check, another subroutine call mechanism will be needed. The irony of these situations is that the "more powerful" mechanism could have been built separately from, but using, simpler mechanisms. Separation would result in a system in which the subset function was available for use where it sufficed.

D. Loops in the "uses" relation.

In many software design projects, the decisions about what other component programs to use are left to individual systems programmers. If a programmer knows of a program in another module, and feels that it would be useful in his program, he includes a call on that program in his text. Programmers are encouraged to use the work of other programmers as much as possible because, when each programmer writes his own routines to perform common functions, we end up with a system that is much larger than it need be.

Unfortunately, there are two sides to the question of program usage. Unless some restraint is exercised, one may end up with a system in which nothing works until everything works. For example, while it may seem wise to have an operating system scheduler use the file system to store its data (rather than use its own disk routines), the result will be that the file system must be present and working before any task scheduling is possible. There are users for whom an operating system subset without a file system would be useful. Even if one has no such users, the subset would be useful during development and testing.

IV. STEPS TOWARDS A BETTER STRUCTURE.

This section discusses four parts of a methodology that I believe will help the software engineer to build systems that do not evidence the problems discussed above.

A. Requirements definition: identifying the subsets first.

One of the clearest morals in the earlier discussion about "design for change" as it is taught in other areas of engineering is that one must anticipate changes before one begins the design. At a recent conference [3], many of the papers exhorted the audience to spend more time identifying the actual requirements before starting on a design. I don't want to repeat such exhortations, but I do want to point out that the identification of the possible subsets is part of identifying the requirements. Treating the easy availability of certain subsets as an operational requirement is especially important to government officials who purchase software. Many officials despair of placing strict controls on the production methods used by their contractors because they are forbidden by law to tell the contractor how to perform his job. They may tell him what they require, but not how to build it. Fortunately, the availability of subsets may be construed as an operational property of the software.

On the other hand, the identification of the required subsets is not a simple matter of asking potential users what they could do without. First, users tend to overstate their requirements. Second, the answer will not characterize the set of subsets that might be wanted in the future. In my experience, identification of the potentially desirable subsets is a demanding intellectual exercise in which one first searches for the minimal subset that might conceivably perform a useful service and then searches for a set of minimal increments to the system. Each increment is small - sometimes so small that it seems trivial. The emphasis on minimality stems from our desire to avoid components that perform more than one function as discussed in section III.C. Identifying the minimal subset is difficult because the minimal system is not usually a program one that anyone would ask for. If we are going to build the software family, the minimal subset is useful, but it is not usually worth building by itself. Similarly, the maximum flexibility is obtained by looking for the smallest possible increments in capability; often these are smaller increments than a user would think of. Whether or not he would think of them before system development, he is likely to want that flexibility later.

The search for a minimal subset and minimal extensions can best be shown by an example. One example of a minimal subset is given in [4]. Another example will be given later in this paper.

B. Information hiding: interface and module definition.

In an earlier section we touched upon the difference between the mathematician's concept of generality and an engineer's approach to design flexibility. Where the mathematician wants his product, a theorem or method of proof, to be as general as possible, i.e applicable, without change, in as many situations as possible, an engineer often must tailor his product to the situation actually at hand. Lack of generality is necessary to make the program as efficient or inexpensive as possible. If he must develop a family of products, he tries to isolate the changeable parts in modules and to develop an interface between the module and the rest of the product that remains valid for all versions. The crucial steps are:

a. Identification of the items that are likely to change. These items are termed "secrets."
b. Location of the specialized components in separate modules.
c. Designing intermodule interfaces that are insensitive to the anticipated the changes. The changeable aspects are termed the "secrets" of the modules.

It is exactly this that the concept of information hiding [5], encapsulation, or abstraction [6] is intended to do for software. Because software is an abstract or mathematical product, the modules may not have any easily recognized physical identity. They are not necessarily separately compilable or coincident with memory overlay units. The interface must be general but the contents should not be. Specialization is necessary for economy and efficiency.

The concept of information hiding is very general and is applicable in many of software change situations - not just the issue of subsets and extensions that we address in this paper. The ideas have also been extensively discussed in the literature [5,6,7,8,9]. The special implications for our problem are simply that, as far as possible, even the presence or absence of a component should be hidden from other components. If one program uses another directly, the presence of the second program cannot be fully hidden from its user. However, there is never any reason for a component to "know" how many other programs use it. All data structures that reveal the presence or number of certain components should be included in separate information hiding modules with abstract interfaces [10]. Space and other considerations make it impossible to discuss this concept further in this paper; it will be illustrated in the example. Readers for whom this concept is new are advised to read some of the articles mentioned above.

C. The virtual machine concept.

To avoid the problems that we have described as "a chain of data transforming components," it is necessary to stop thinking of systems in terms of components that correspond to steps in the processing. This way of thinking dies hard. It is almost certain that your first introduction to programming was in terms of a series of statements intended to be executed in the order that they were explained to you. We are goal oriented; we know what we start with and what we want to produce. It is natural to think in terms of steps progressing towards that goal. It is the fact that we are designing a family of systems that makes this "natural" approach the wrong one.

The viewpoint that seems most appropriate to designing software families is often termed the virtual machine approach. Rather than write programs that perform the transformation from input data to output data, we design software machine extensions that will be useful in writing many such programs. Where our hardware machine provides us with a set of instructions that operate on a small set of data types, the extended or virtual machine will have additional data types as well as "software instructions" that operate on those data types. These added features will be tailored to the class of programs that we are building. While the VM instructions are designed to be generally useful, they can be left out of a final product if the user's programs don't use them. The programmer writing programs for the virtual machine should not need to distinguish between instructions that are implemented in software and those that are hardware implemented. To achieve a true virtual machine, the hardware resources that are used in implementing the extended instruction set must be unavailable to the user of the virtual machine. He has traded these resources for the new data elements and instructions. Any attempt to use those resources again will invalidate the concept of virtual machine and lead to complications. Failure to provide for isolation of resources is one of the reasons for the failure of some attempts to use macros to provide a virtual machine. The macro user must be careful not to use the resources used in the macros.

There is no reason to accomplish the transformation from the hardware machine to a virtual machine with all of the desired features in a single leap. Instead we will use the machine at hand to implement a few new instructions. At each step we take advantage of the newly introduced features. Such a step-by-step approach turns a large problem into a set of small ones and, as we will see later, eases the problem of finding the appropriate subsets. Each element in this series of virtual machines is a useful subset of the system.

D. Designing the "uses" structure.

The concept of an abstract machine is an intuitive way of thinking about design. A precise description of the concept comes through a discussion of the relation "uses" [11,12].

1. The relation "uses."

We consider a system to be divided into a set of programs that can be invoked either by the normal flow of control mechanisms, by an interrupt, or by an exception handling mechanism. Each of these programs is assumed to have a specification that defines exactly the effect that an invocation of the program should have.

We say of two programs A and B that A <u>uses</u> B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A <u>uses</u> B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Note that to decide whether A <u>uses</u> B or not, one must examine <u>both</u> the implementation <u>and</u> the specification of A.

The "<u>uses</u>" relation and "invokes" very often coincide, but <u>uses</u> differs from <u>invokes</u> in two ways:

(1) Certain invocations may not be instances of "<u>uses</u>." If A's specification requires only that A <u>invoke</u> B when certain conditions occur, then A has fulfilled its specification when it has generated a correct call to B. A is correct even if B is incorrect or absent. A proof of correctness of A need only make assumptions about the way to invoke B.

(2) A program A may use B even though it never invokes it. The best illustration of this is interrupt handling. Most programs in a computer system are only correct on the assumption that the interrupt handling routine will correctly handle the interrupts (leave the processor in an acceptable state). Such programs use the interrupt handling routines even though they never call them. "<u>Uses</u>" can be more precisely formulated as "<u>requires the presence of a correct version of</u>."

Systems that have achieved a certain "elegance" (e.g., T.H.E. [5], Venus [6]) have done so by having parts of the system "<u>use</u>" other parts in such a way that the "user" programs were simplified. For example, the transput stream mechanism in T.H.E. <u>uses</u> the segmenting mechanism to great advantage. In contrast, many large and complex operating systems achieve their size and complexity by having "independent" parts. For example, there are many systems in which "spooling," virtual memory management, and the file system all perform their own backup store operations. Code to perform these functions is present in each of the components. Whenever such components must share a single device, complex interfaces exist.

The disadvantage of unrestrained "usage" of each others facilities is that the system parts become highly interdependent. Often there are no subsets of the system that can be used before the whole system is complete. In practice, some duplication of effort seems preferable to a system in which nothing runs unless everything runs.

2. The uses hierarchy.

By restricting the relation "<u>uses</u>" so that its graph is loop free we can retain the primary advantages of having system parts "<u>use</u>" each other while eliminating the problems. In that case it is possible to assign the programs to the levels of a hierarchy by the following rules:

1. Level 0 is the set of all programs that <u>use</u> no other program.
2. Level i is the set of all programs that <u>use</u> at least one program on level i-1 and no program at a level higher than i-1.

If such a hierarchical ordering exists, then each level offers a testable and usable subset of the system. In fact, one can get additional subsets by including only parts of a level. This property is very valuable for the construction of any software system and is vital for developing a <u>broad</u> family of systems.

The design of the "uses" hierarchy should be one of the major milestones in a design effort. The division of the system into independently callable subprograms has to go on in parallel with the decisions about <u>uses</u>, because they influence each other.

3. The criteria to be used in allowing one program to use another.

We propose to allow A "<u>uses</u>" B when all of the following conditions hold:

(a) A is essentially simpler because it uses B.

(b) B is not substantially more complex because it is not allowed to use A.

(c) There is a useful subset containing B and not needing A.

(d) There is no conceivably useful subset containing A but not B.

During the process of designing the "uses" relation, we often find ourselves in a situation where two programs could obviously benefit from using each other and the conditions above cannot be satisfied. In such situations, we resolve the apparent conflicts by a technique that we call "sandwiching." One of the programs is "sliced" into two parts in a way that allows the programs to "use" each other and still satisfy the above conditions. If we find ourselves in a position where A would benefit from using B, but B can also benefit from using A, we may split B into two programs: B1 and B2. We then allow A to use B2 and B1 to use A. The result would appear to be a sandwich with B as the bread and A as the filling. Often, we then go on to split A. We start with a few levels and end up with many.

The most frequent instances of splitting and sandwiching came because initially we were assuming that a "level" would be a "module" in the sense of IV. B. We will discuss this in the final part of this paper.

4. Use of the word "convenience."

It will trouble some readers that it is usual to use the word "convenience" to describe a reason for introducing a certain facility at a given level of the hierarchy. A more substantial basis would seem more scientific.

As discussed in [11] and [13], we must assume that the hardware itself is capable of performing all necessary functions. As one goes higher in the levels, one can lose capabilities (as resouces are consumed) - not gain them. On the other hand, at the higher levels the new functions can be implemented with simpler programs because of the additional programs that can be used. We speak of "convenience" to make it clear that one could implement any functions on a lower level, but the availability of the additional programs at the higher level is useful. For each function we give the lowest level at which the features that are useful for implementing that function (with the stated restrictions), are available. In each case, we see no functions available at the next higher level that would be useful for implementing the functions as described. If we implemented the program one level lower we would have to duplicate programs that become available at that level.

V. EXAMPLE: AN ADDRESS PROCESSING SUBSYSTEM

As an example of designing for extensibility and subsets, we consider a set of programs to read in, store, and write out lists of addresses. This example has also been used, to illustrate a different point, in [10] and has been used in several classroom experiments to demonstrate module interchangeability.

A. Our Basic Assumptions Are:

1. The information items discussed in Figure 1 will be the items to be processed by all application programs.

2. The input formats of the addresses are subject to change.

3. The output formats of the addresses are subject to change.

4. Some systems will use a single fixed format for input and output. Other systems will need the ability to choose from several of input or output formats at run-time. Some systems will be required in which the user can specify the format using an address format definition language.

5. The representation of addresses in main storage will vary from system to system.

6. In most systems, only a subset of the total set of addresses stored in the system need be in main storage at any one time. The number of addresses needed may vary from system to system and, in some systems the number of addresses to be kept in main memory may vary at run-time.

The following items of information will be found in the addresses to be processed and constitute the only items of relevance to the application programs:

- Last name
- Given names (first name and possible middle names)
- Organization (Command or Activity)
- Internal identifier (Branch or Code)
- Street address or P.O. box
- City or mail unit identifier
- State
- Zip code
- Title
- Branch of service if military
- GS grade if civil service

Each of the above will be strings of characters in the standard ANSI alphabet, and each of the above may be empty or blank.

FIGURE I

B. We Propose the Following Design Decisions:

1. The input and output programs will be table driven; the table will specify the format to be used for input and output. The contents and organization of these format tables will be the 'secrets' of the input and output modules.

2. The representation of addresses in core will be the 'secret' of an Address Storage Module (ASM). The implementation chosen for this module will be such that the operations of changing a portion of an address will be relatively inexpensive, compared to making the address table larger or smaller.

3. When the number of addresses to be stored exceeds the capacity of an ASM, programs will use an Address File Module (AFM). An AFM can be made upward compatible with an ASM; programs that were written to use ASM's could operate using an AFM in the same way. The AFM provides additional commands to allow more efficient usage by programs that do not assume the random access properties of an ASM. These programs are described below.

4. Our implementaton of an AFM would use an ASM as a submodule as well as another submodule that we will call Block File Module (BFM). The BFM stores blocks of data that are sufficiently large to represent an address, but the BFM is not specialized to the handling of addresses. An ASM that is used within an AFM may be said to have two interfaces. In the "normal interface" that an ASM presents to an outside user, an address is a set of fields and the access functions hide or abstract from the representation. Figure 2 is a list of the access programs that comprise this interface. In the second interface, the ASM deals with blocks of contiguous storage and abstracts from the contents. There are commands for the ASM to input and output 'addresses' but the operands are storage blocks whose interpretation as addresses is known only within the ASM. The AFM makes assumptions about the association between blocks and addresses but not about the way that an address's components are represented as blocks. The BFM is completely independent of the fact that the blocks contain address information. The BFM might, in fact, be a manufacturer supplied access method.

ACCESS FUNCTIONS FOR "NORMAL INTERFACE"

MODULE: ASM

NAME OF ACCESS PROGRAM*	INPUT PARAMETERS						OUTPUT	
*ADDTIT:	asm	X	integer	X	string	→	asm	•
ADDGN:	asm	X	integer	X	string	→	asm	•
ADDLN:	asm	X	integer	X	string	→	asm	•
ADDSERV:	asm	X	integer	X	string	→	asm	•
ADDBORC:	asm	X	integer	X	string	→	asm	•
ADDCORA:	asm	X	integer	X	string	→	asm	•
ADDSORP:	asm	X	integer	X	string	→	asm	•
ADDCITY:	asm	X	integer	X	string	→	asm	•
ADDSTATE:	asm	X	integer	X	string	→	asm	•
ADDZIP:	asm	X	integer	X	string	→	asm	•
ADDGSL:	asm	X	integer	X	string	→	asm	•
SETNUM:	asm	X	integer	→	asm •			
FETTIT:	asm	X	integer	→	string			
FETGN:	asm	X	integer	→	string			
FETGN:	asm	X	integer	→	string			
FETLN:	asm	X	integer	→	string			
FETSERV:	asm	X	integer	→	string			
FETBORC:	asm	X	integer	→	string			
FETCORA:	asm	X	integer	→	string			
FETSORP:	asm	X	integer	→	string			
FETCITY:	asm	X	integer	→	string			
FETSTATE:	asm	X	integer	→	string			
FETZIP:	asm	X	integer	→	string			
FETGSL:	asm	X	integer	→	string			
FETNUM:	asm	→	integer					

FIGURE 2 - SYNTAX OF ASM FUNCTIONS

*These are abreviations: ADDTIT = ADD TITLE; ADDGN = ADD GIVEN NAME, etc.

C. <u>Component Programs</u>.

1. Module: Address Input

INAD:	Reads in an address that is assumed to be in a format specified by a format table and calls ASM or AFM functions to store it.
INFSL:	Selects a format from an existing set of format tables. The selected format is the one that will be used by INAD. There is always a format selected.
INFCR:	Adds a new format to the tables used by INFSL. The format is specified in a 'format language.' Selection is <u>not</u> changed (i.e., INAD still uses the same format table).
INTABEXT:	Adds a blank table to the set of input format tables.
INTABCHG:	Rewrites a table in the input format tables using a description in a format language. Selection is not changed.
INFDEL:	Deletes a table from the set of format tables. The selected format cannot be deleted.
INADSEL:	Reads in an address using one of a set of formats. Choice is specified by an integer parameter.
INADFO:	Reads in an address in a format specified as one of its parameters (a string in the format definition language). The format is selected and added to the tables and subsequent addresses could be read in using INAD.

2. Module: Address Output

OUTAD:	Prints an address in a format specified by a format table. The information to be printed is assumed to be in an ASM and identified by its position in an ASM.
OUTFSL:	Selects a format table from an existing set of output format tables. The selected FORMAT is the one that will be used by OUTAD.
OUTTABEXT:	Adds a "blank" table to the set of output format tables.
OUTTABCHG:	Rewrites the contents of a format table using information in a format language.
OUTFCR:	Adds a new format to the set of formats that can be selected by OUTFSL in a format description language.
OUTFDEL:	Deletes a table from the set of FORMAT tables that can be selected by OUTFSL.
OUTADSEL:	Prints out an address using one of a set of formats.
OUTADFO:	Prints out an address in a format specified in a format definition language string, which is one of the actual parameters. The format is added to the tables and selected.

3. Module: Address Storage (ASM)

FET (Component Name):
This is a set of functions used to read information from an address store. Returns a string as a value. See Figure 2.

ADD (Component Name):
This is a set of functions used to write information in an address store. Each takes a string and an integer as parameters. The integer specifies an address within the ASM. See Figure 2.

OBLOCK:	Takes an integer parameter, returns a storage block as a value.
IBLOCK:	Accepts a storage block and integer as parameters. Its effect is to change the contents of an address store - which is reflected by a change in the values of the FET programs.
ASMEXT:	Extends an address store by appending a new address with empty components at the end of the address store.

ASMSHR:	"Shrinks" the address store.
ASMCR:	Creates a new address store. The parameter specifies the number of components. All components are initially empty.
ASMDEL:	Deletes an existing address store.

4. Module: Block File Module

BLFET:	Accepts an integer as a parameter and returns a "block."
BLSTO:	Accepts a block and an integer and stores the block.
BFEXT:	Extends BFM by adding additional blocks to its capacity.
BFSHR:	Reduces the size of the BFM by removing some blocks.
BFMCR:	Creates a files of blocks.
BFMDEL:	Deletes an existing file of blocks.

5. Module: Address File Module

This modules includes implementations of all of the ASM functions except OBLOCK and IBLOCK. To avoid confusion in the diagram showing the uses hierarchy we have changed the names to:

AFMADD(Component Name) defined as in Figure 2
AFMFET(Component Name) defined as in Figure 1
AFMEXT defined as in BFM above
AMFSHR defined as in BFM above
AFMCR defined as in BFM above
AFMDEL defined as in BFM above

D. <u>Uses Relation</u>

Figure 3 shows the <u>uses</u> relation between the component programs. It is important to note that we are now discussing the implementation of those programs, not just their specification. The <u>uses</u> relation is characterized by the fact that there are a large number of relatively simple, <u>single purpose</u> programs on the lowest level. The upper level programs are implemented by means of these lower level programs so that they too are quite simple. This <u>uses</u> relation diagram characterizes the set of possible subsets.

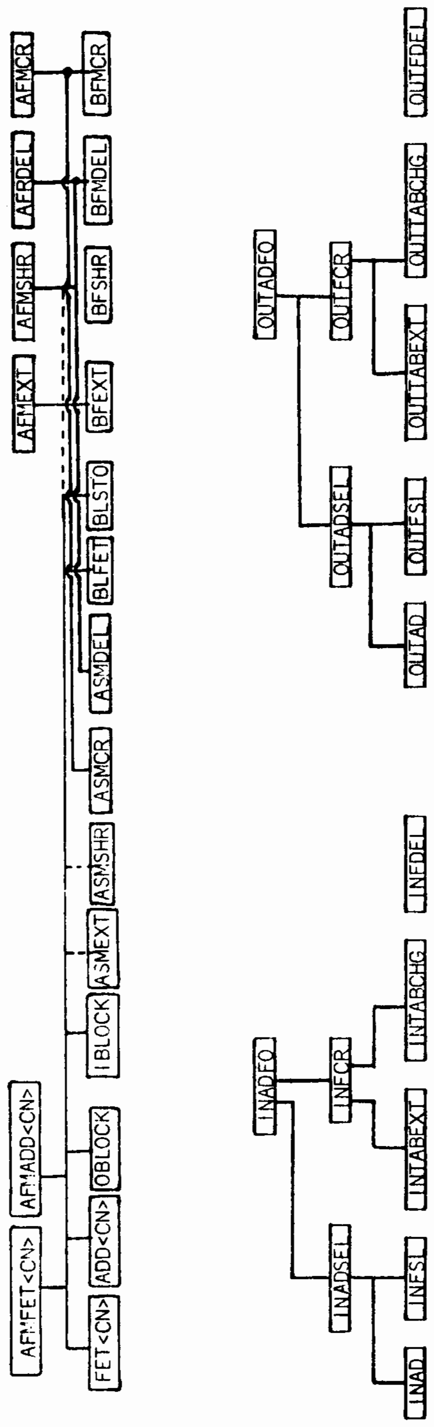

FIGURE 3

E. Discussion

To pick a subset, one identifies the set of upper level programs that the user needs and includes only those programs that those programs use (directly or indirectly). For example, a user who uses addresses in a single format does not need the component programs that interpret format description languages. Systems that work with a small set of addresses can be built without any BFM components. A program that works as a query system and never prints out a complete address would not need any Address Output components.

The system is also easily extended. For example, one could add a capability to read in addresses with self-defining files. If the first record on a file was a description of the format in something equivalent to the format description language, one could write a program that would be able to read in that record, use INTABCHG to build a new format table, and then read in the addresses. Programs that do things with addresses (such as print out "personalized" form letters) can also be added using these programs and selecting only those capabilities that they actually need.

One other observation that can be made is that the upper level programs can be used to "generate" lower level versions. For example, the format description languages can be used to generate the tables used for the fixed format versions. There is no need for a separate SYSGEN program.

We will elaborate on this observation in the conclusion.

X. SOME REMARKS ON OPERATING SYSTEMS: WHY GENERALS ARE SUPERIOR TO COLONELS

An earlier report [11] discusses the design of a "uses" hierarchy for operating systems. Although there have been some refinements to the proposals of that report, its basic contents are consistent with the present proposals. This section compares the approach outlined in this paper and the "kernel" approach or "nucleus" approach to OS design [18,19,20]. It is tempting to say that the suggestions in this paper do not conflict with the "kernel" approach. These proposals can be viewed as a refinement of the nucleus approach. The first few levels of our system could be labeled "kernel," and one could conclude that we are just discussing a fine structure within the kernel.

To yield to that temptation would be to ignore an essential difference between the approaches suggested in this paper and the kernel approach. The system kernels known to me are such that some desirable subsets cannot be obtained without major surgery. It was assumed that the nucleus must be in every system family member. In the RC4000 system the inability to separate synchronization from message passing has led some users to bypass the kernel to perform teletype handling functions. In Hydra as originally proposed [19], "type checking" was so intrinsic to the call mechanism that it appeared impossible to disable it when it was not needed or affordable.*

Drawing a line between "kernel" and the rest of the system, and putting "essential" services of "critical programs" in the nucleus yields a system in which kernel features cannot be removed and certain extensions are impractical. Looking for a minimal subset and a set of minimal independent incremental function leads to a system in which one can trim away unneeded features. I know of no feature that is always needed. When we say that two functions are almost always used together, we should remember that "almost" is a euphemism for "not."

XI. SUMMATION

This paper describes an approach to software intended to result in systems that can be tailored to fit the needs of a broad variety of users. The points most worthy of emphasis are:

1. The Requirements Include Subsets and Extensions.

It is essential to recognize the identification of useable subsets as part of the preliminaries to software design. Flexibility cannot be an afterthought. Subsetability is needed, not just to meet a variety of customers' needs, but to provide a fail-soft way of handling schedule slippage.

2. Advantages of the Virtual Machine Approach.

Designing software as a set of virtual machines has definite advantages over the conventional (flow chart) approach to system design. The virtual machine "instructions" provide facilities that are useful for purposes beyond those originally conceived. These instructions can easily be omitted from a system if

---

*Accurate reports on the current status and performance of that system are not available to me.

they are not needed. Remove a major box from a flow chart and there is often a need to "fill the hole" with conversion programs.

3. On the Difference Between Software Generality and Software Flexibility.

Software can be considered "general" if it can be used, without change, in a variety of situations. Software can be considered flexible, if it is easily changed to be used in a variety of situations. It appears unavoidable that there is a run-time cost to be paid for generality. Clever designers can achieve flexibility without significant run-time cost, but there is a design-time cost. One should incur the design-time cost only if one expects to recover it when changes are made.

Some organizations may choose to pay the run-time cost for generality. They build general software rather than flexible software because of the maintenance problems associated with maintaining several different versions. Factors influencing this decision include (a) the availability of extra computer resources, (b) the facilities for program change and maintenance available at each installation, and (c) the extent to which design techniques ease the task of applying the same change to many versions of a program.

No one can tell a designer how much flexibility and generality should be built into a product, but the decision should be a conscious one. Often, it just happens.

4. On the distinction between modules, subprograms, and levels.

Several systems and at least one dissertation [14,15,16,17] have, in my opinion, blurred the distinction between modules, subprograms and levels. Conventional programming techniques consider a subroutine or other callable program to be a module. If one wants the modules to include all programs that must be designed together and changed together, then, as our example illustrates, one will usually include many small subprograms in a single module. It doesn't matter what word we use; the point is that the unit of change is not a single callable subprogram.

In several systems, modules and levels have coincided [14,15]. This had led to the phrase "level of abstraction." Each of the modules in the example abstract from some detail that is assumed likely to change. However, there is no correspondence between modules and levels. Further, I have not found a relation, "more abstract than," that would allow me to define an abstraction hierarchy [12]. Although I am myself guilty of using it, in most cases the phrase "levels of abstraction" is an abuse of language.

Janson has suggested that a design such as this one (or the one discussed in [11]) contain "soft modules" that can represent a breach of security principles. Obviously an error in any program in one of our modules can violate the integrity of that module. All module programs that will be included in a given subset must be considered in proving the correctness of that module. However, I see no way that allowing the component programs to be on different levels of a "uses" hierarchy makes this process more difficult or makes the system less secure. The boundaries of our modules are quite firm and clearly identified.

The essential difference between this paper and other discussions of hierarchically structured designs is the emphasis on subsets and extensions. My search for a criterium to be used in designing the uses hierarchy has convinced me that if one does not care about the existence of subsets, it doesn't really matter what hierarchy one uses. Any design can be bent until it works. It is only in the ease of decomposition that they differ.

5. On Avoiding Duplication.

Some earlier work [21] has suggested that one needs to have duplicate or near duplicate modules in a hierarchically structured system. For example, they suggest that one needs one implementation of processes to give a fixed number of processes at a low level and another to provide for a varying number of processes at a user's level. Similar ideas have appeared elsewhere. Were such duplication to be necessary, it would be a sound argument against the use of "structured" approaches. One can avoid such duplication if one allows the programs that vary the size of a data structure to be on a higher level than the other programs that operate on that data structure. For example, in an operating system, the programs to create and delete processes need not be on the same level as the more frequently used scheduling operations. In designing software, I regard the need to perform similar functions in two programs as an indication of a fundamental error in my thinking.

6. Designing for Subsets and Extensions can Reduce the Need for Support Software.

We have already mentioned that this design approach can eliminate the need for separate SYSGEN programs. We can also eliminate the need for special purpose compilers. The price of the convenience features offered by such languages is often a compiler and run-time package distinctly larger than the system being built. In our approach, each level provides a language extention available to the programmers of the next level. We never build a compiler; we just build our system, but we get convenience features anyway.

7. Extension at Run-Time Vs. Extension During SYSGEN.

At a later stage in the design we will have to choose data structures and take the difference between run-time extension and SYSGEN extension into consideration. Certain data structures are more easily accessed but harder to extend while the program is running; others are easily extended but at the expense of a higher access cost. These differences do not affect our early design decisions because they are hidden in modules.

8. On the Value of a Model.

My work on this example and similar ones has gone much faster because I have learned to exploit a pattern that I first noticed in the design discussed in [11]. Low level operations assume the existence of a fixed data structure of some type. The operations on the next level allow the swapping of a data element with others from a fixed set of similar elements. The high level programs allow the creation and deletion of such data elements. This pattern appears several times in both designs. Although I have not designed your system for you, I believe that you can take advantage of a similar pattern. If so, this paper has served its purpose.

ACKNOWLEDGMENTS

The ideas presented in this paper have been developed over a lengthy period and with the cooperation and help of many collaborators. The earliest work was supported by NV Philips Computer Industrie, Apeldoorn, The Netherlands, and I am grateful to numerous Philips employees for thought provoking comments and questions. William Price's collaboration and NSF support were invaluable at Carnegie-Mellow University. The support of the German Federal Ministry for Research and Technology (BMFT) and the help of W. Bartussek, G. Handzel, and H. Wuerges at the Technische Hochschule Darmstadt led to substantial improvements. Kathryn Heninger, David Weiss, and John Shore at the Naval Research Laboratory helped me to understand the application of the concepts in areas other than operating systems. Barbara Trombka and John Guttag both helped in the design of pilots of the address process system. Discussions with P. J. Courtois have helped me to better understand the relation between software structure and run-time characteristics of computer systems. Dr. Edward Britton, Mr. H. Rettenmaier, Mr. Laslo Belady, Dr. Donald Stanat, G. Frank, and Dr. William Wright made many helpful suggestions about an earlier draft of this paper. If you find portions of this paper helpful, these people deserve your thanks.

REFERENCES

[1] Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, 1976.

[2] Parnas,D.L. "On the Design and Development of Program Families." IEEE Transactions on Software Engineering, March 1976.

[3] 2nd International Conference on Software Engineering, 13-15 October 1976; Special issue of IEEE Transactions on Software Engineering, December 1976.

[4] Parnas, D.L., Handzel, G., and H. Wuerges. "Design and Specification of the Minimal Subset of an Operating System Family." Presented at 2nd International Conference on Software Engineering, 13-15 October 1976; published in special issue of IEEE Transactions on Software Engineering, December 1976.

[5] Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." Comm. ACM, December 1972.

[6] Linden, T.A. "The Use of Abstract Data Types to Simplify Program Modifications." Proceedings of Conference on Data: Abstraction, Definition and Structure, March 22-24, 1976; published in ACM SIGPLAN Notices, Vol. II, 1976 Special Issue.

[7] Parnas,D.L. "A Technique for Software Module Specification with Examples." Comm. ACM, May 1972.

[8] Parnas, D.L. "Information Distribution Aspects of Design Methodology." Proc. IFIP Congress, 1971.

[9] Parnas, D.L. "The Use of Precise Specifications in the Development of Software." Proc. IFIP Congress, 1977, North Holland Publishing Company.

[10] Parnas, D.L. "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems." NRL Report 8047, Naval Research Laboratory, Washington, D.C., June 1977.

[11] Parnas, D.L. "Some Hypotheses About the 'Uses' Hierarchy for Operating Systems." Technical Report, Technische Hochschule Darmstadt, Darmstadt, West Germany, March 1976.

[12] Parnas, D.L. "On a 'Buzzword': Hierarchical Structure." Proc. IFIP Congress, 1974, North Holland Publishing Company, 1974.

[13] Parnas, D.L. and D.L. Siewiorek. "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems." Comm. ACM, 18 (7), July 1975.

[14] Dijkstra, E.W. "The Structure of the "THE"-Multiprogramming System." CACM, 11, 5 (May 1968), pp. 341-346.

[15] Liskov, B. "The Design of the Venus Operating System. CACM, 15, 3 (March 1972), pp. 144-149.

[16] Janson, P.A. "Using Type Extension to Organize Virtual Memory Mechanisms." MIT-LCS-TR-167, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., September 1976.

[17] Janson, P.A. "Using Type-Extension to Organize Virtual Memory Mechanisms." Research Report RZ 858 (#28909) 8/31/77, IBM Zurich Research Laboratory, Switzerland.

[18] Brinch-Hansen, P. "The Nucleus of the Multiprogramming System." CACM, 13, 4 (April 1970), pp. 238-241, 250.

[19] Wulf, W., Cohen, E., Jones, A., Lewin, R., Pierson, C., and F. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System." CACM, 17, 6 (June 1974), pp. 337-345.

[20] Popek, G.J. and C.S. Kline. "The Design of a Verified Protection System." Proc. Intl. Workshop on Prot. in Oper. Syst., IRIA, pp. 1 183-196.

[21] Saxena, A.R. and T.H. Bredt. "A Structured Specification of a Hierarchical Operating System." Proceedings of the 1975 International Conference on Reliable Software.