CHAPTER 6

Program Structure Design

The next software design process is the design of the program structure, which includes the definition of all modules in the program, the hierarchical structure of the modules, and the interfaces among the modules. If the product being designed is a single program, the input to this process is the detailed external specification. If the product is a system, the input is the detailed external specification and the system architecture, and this process is the structural design of all components or subsystems in the system.

The traditional method of managing complexity is the idea of "divide and rule," often called "modularization." However, in practice this idea has often been ineffective in reducing complexity. Liskov [1] has pointed out three reasons for this failure:

1. Modules are made to do too many related but different functions, obscuring their logic.
2. Common functions are not identified in the design, resulting in their distribution (and varied implementation) among many different modules.
3. Modules interact on shared or common data in unexpected ways.

A design methodology called *composite design* [2] is the design principle discussed here for program structure design. Composite design actually consists of two sets of principles: a set of explicit design measures that solve the three problems listed above plus many additional problems, and a set of thought processes for decomposing a program into a set of modules, module interfaces, and module relationships. Composite design leads to a program structure of minimal complexity, which has proven to increase reliability, maintainability, and adaptability.

MODULE INDEPENDENCE

The primary way to make a program less complex is to decompose it into a large set of small, highly independent modules. A module is a closed subroutine that can be called from any other module in the program and can be separately compiled (note that this excludes the PL/I internal procedure and the COBOL "performed" paragraph). High independence can be achieved by two optimization methods: maximizing the relationships within each module and minimizing the relationships among modules. Given that a program will eventually consist of a set of program statements where the statements will have some relationships among themselves (both in terms of function performed and data manipulated), what is needed is the foresight to organize these statements into separate boxes (modules) such that the statements within any module are closely related and any pair of statements in two different modules are minimally related. The goals are to isolate a single function to each module (high module strength) and to minimize the data relationships among modules by using formal parameter-passing methods (loose module coupling).

MODULE STRENGTH

Module strength is a measure of the relationships within a module. Determining the strength of a module involves analyzing the function or functions performed by the module and then fitting the module into one of seven categories. The categories were established partially to quantify the "goodness" of particular types of modules.

Before proceeding it is necessary to define what is meant by the *function* of a module. A module has three basic attributes: it performs one or more *functions*, it contains some *logic*, and it is used in one or more *contexts*. Function is an external description of the module; it describes what the module does when it is called, but not *how* it does it. Logic describes the internal algorithm of the module, in other words, how it performs its function. Context describes a particular usage of a module. For instance, a module with the function "squeeze the blanks from a character string" might be used in the context "compress teleprocessing message." To see the difference between function and logic, consider a module with the function: compile a PL/I program. This module may be the top module in an 83-module compiler or it may be the only module in the compiler. In either case its function remains the same but its logic is much different. Therefore, the function of a module can be viewed as the summation of the module's logic plus the functions of all subordinate (called) modules. This definition is recursive and applies to any module in a hierarchy.

The design goal is to define modules such that each module performs one function (such modules are said to have *functional strength*). To understand the need for this goal, the scale for the seven categories of module strength is explored as follows, starting with the weakest type of strength.

A *coincidental-strength* module is a module in which there are no meaningful relationships among its elements. It is difficult to give an example of such a module since it performs no meaningful function. The only way to describe the module is by describing its logic. One way a module of this type might occur is in an after-the-fact "modularization" of a program, where we discover identical sequences of code in several modules and decide to group these together into a single module. If these code sequences (although appearing to be identical) have different meanings in the modules in which they originally appeared, this new module has coincidental strength. A module of this sort is closely related to its calling modules, implying that almost any modification of the module on behalf of one of its callers will cause it to operate incorrectly for its other callers.

A *logical-strength* module is a module that, during each invocation, performs one selected function from a class of related functions. The selected function is explicitly requested by the calling module, for instance by the use of a function code. An example is a module whose function is to read or write a record to/from a file. The main problem with this type of module is the use of a single interface to reflect multiple functions. This leads to complex interfaces and unexpected errors in instances where the interface is modified to reflect a change in one of the functions.

A *classical-strength* module is a module that sequentially performs a class of related functions. The most common examples are "initialization" and "termination" modules. The major problem with modules of this type is that they usually have implicit relationships with other modules in the program, making the program difficult to understand and leading to errors when the program must be modified.

A *procedural-strength* module is a module that sequentially performs a class of related functions, where the functions are related in terms of the procedure of the problem. The "problem" is the reason for writing the program. An example of a problem is: write a program to regulate the temperature of the primary boiler. Problems to a certain extent dictate the procedure of the program. For instance, the problem might state that upon receiving signal x, valve y should be turned off and the temperature should be read and logged. A module whose function is "turn off valve y, read boiler temperature and record it on journal" has procedural strength. The only reliability problem is that the code for the set of functions may be intertwined. Note that this type of module, in common with most other types, has other non-reliability-related problems, as shown by Myers [2].

A *communicational-strength* module is a module with procedural strength but with one additional relationship: *all* of its functions are related in terms of data usage. For instance, the module "read next transaction and update master file" has communicational strength since both functions are related by their use of the transaction. Again, the functions may tend to be intertwined, but the risk of making a mistake while performing a modification is somewhat less since the functions are more closely related.

Informational strength is next on the scale. However, I defer discussion of it for a moment.

A *functional-strength* module is a module that performs a single specific function such as "turn off valve y," "execute EDIT command," or "summarize the week's transactions." Functional strength is the highest (best) form of module strength.

Note that a functional-strength module could also be described as a set of more-detailed functions. For instance, a "summarize transaction tape" module could have been described as "initialize summary table, open transaction file, read records, and update summary table." The reader may look at this and have the feeling that by just rewording the module's description its strength has been lowered. The resolution is that if these "lower functions" can rationally be described as a single "higher" well-defined function, the module has functional strength.

The remaining type of strength is *informational strength*. An informational-strength module is a module that performs several functions, where the functions operate on the same data structure and each function is represented by a unique entry point. A module with two entry points, one having the function "insert entry into symbol table" and the other having the function "search symbol table," has informational strength. This type of module can be viewed as the physical grouping of certain functional-strength modules to achieve "information hiding" [3], such as hiding all knowledge of a particular data structure, resource, or device to within a single module. In the example mentioned, all knowledge of the symbol table structure and location are hidden within one module. The advantage in this is that whenever some aspect of the program can be hidden within a single module, the independence among the program's modules increases. The design goal mentioned earlier is now modified to include informational-strength modules, as well as functional-strength modules, as the goal.

Although the preceding discussion focused only on the relationship between module strength and susceptibility to errors, module strength also affects the adaptability of the program, the difficulty of testing individual modules, and the degree to which a module is usable in other contexts and other programs [2]. The strength scale was ordered by weighing all of these attributes.

Note that a module may fit the descriptions of several types of strength. For instance, a communicational-strength module also fits the definition of procedural and classical strength. A module is always classified as having the highest strength whose definition it meets.

MODULE COUPLING

The second primary way to maximize module independence is by minimizing the connections among modules. Module coupling, a measure of the data relationships among modules, is concerned with both the mechanism used to pass data and the attributes of the data itself. Every pair of modules in a program can be analyzed and fit into either one of six categories of coupling or else be categorized as having no direct coupling.

The design goal is to define module interfaces so that all data passed between modules is in the form of explicit simple parameters. Again, to understand the importance of this goal, the six categories of coupling are explored below, starting with the tightest form of coupling (the worst case).

Two modules are *content coupled* if one *directly* references the

contents of the other. For instance, if module A somehow references data in module B by using an absolute displacement, the modules are content coupled. Almost any change to B, or maybe just recompiling B with a different version of the compiler, will introduce an error into the program. Fortunately, most high-level languages make content coupling difficult to achieve.

A group of modules are *common coupled* if they reference the same global data structure. A set of PL/I modules that reference a data structure declared as EXTERNAL are common coupled to one another. FORTRAN modules referencing data in a COMMON area and groups of modules referencing a data structure in an absolute storage location (including registers) are also examples of common coupling.

There are a large number of problems associated with common coupling. All of the modules are dependent on the physical ordering of the items within the structure, implying that a change to the size of one data item affects all of the modules. Use of global data defeats attempts to control the access that each module has to data. For example, IBM's OS/360 has a large global data structure called the communications vector table. The inability to control access to this table (and other global tables) has led to a number of reliability and adaptability problems. Global variable names bind modules together when they are originally coded. This means that the reuse of common-coupled modules in future programs is difficult if not impossible.

The use of global data also reduces a program's readability. Consider the following piece of a program:

```
DO WHILE (A);
     CALL L (X,Y,Z);
     CALL M (X,Y);
     CALL N (W,Z);
     CALL P (Z,X,Y);
END;
```

If A is not a global variable and if other bad coding practices are avoided (such as overlaying A and W, X, Y, or Z), we can state that the loop cannot terminate. If A is a global variable, we cannot immediately determine if the loop can terminate. We have to explore the insides of modules L, M, N, and P, and also the insides of all modules called by these four modules, to understand the DO loop!

The case against global data is becoming as important as the case against the GO TO statement and is beginning to receive attention in the professional literature [4, 5, 6].

A group of modules are *external coupled* if they reference the same global data item (single-field variable). For instance, a set of PL/I modules referencing a variable (not a structure) declared as EXTERNAL are external coupled with one another. External coupling has many of the problems associated with common coupling. However, the problem of dependence on the physical ordering of items within a structure is not present in external coupling.

Two modules are *control coupled* if one explicitly controls the functions of the other, for instance by the use of a function code. Control coupling and logical strength usually occur together; thus the major problem here is the same one as associated with logical strength: the use of a single complex interface to reflect one of many functions. Control coupling also often implies that the calling module has some knowledge of the logic of the called module, thus lessening their independence.

A group of modules are *stamp coupled* if they reference the same nonglobal data structure. If module A calls module B passing B an employee personnel record and both A and B are sensitive to the structure or format of the record, then A and B are stamp coupled.

Stamp coupling should be avoided where possible because it creates unnecessary connections between modules. Suppose module B only needs a few fields in the personnel record. By passing it the entire record, B is forced to be aware of the entire structure of the record and the chances of module B's inadvertently modifying the record are increased. (It seems fair to say that the more extraneous data to which a module is exposed, the greater the opportunity for error.)

Stamp coupling can often be eliminated by isolating all functions performed on a particular data structure to an informational-strength module. Other modules may need to name the structure, but they know only its name (address), not its format. This technique is illustrated later in this chapter.

Two modules are *data coupled* if one calls the other and all inputs to, and outputs from, the called module are data item parameters (not structures). Suppose in the example above that module B's function is to print an envelope for an employee. Rather than giving B the personnel record, we could pass it the employee's name, street, apartment, city, state, and zip code as arguments. Module B is now not dependent on the personnel record. A and B are more independent and the probability of an error in B is lower since the programmer of B has less data to deal with.

As was the case for module strength, module coupling affects other attributes that have not been discussed, such as adaptability, the difficulty of testing modules, the reusability of modules, and the ease or difficulty of multiprogramming [2].

A pair of modules can fit the definition of several types of coupling. For instance, two modules could be both stamp coupled and external coupled. When this occurs the modules are defined to have the tightest (worst) type of coupling that they exhibit (in this case external coupling).

The measures of strength and coupling can be used to evaluate an existing design or as guidelines in producing a design for a new program. They should not imply, however, that a design with instances of strengths and couplings below the ideal is necessarily a poor design. The strength and coupling measures are guidelines. A designer may decide, based on some tradeoff, to define a logical-strength module. However, when he does this, he should be able objectively to explain his reasons and also to realize the implications of his tradeoff. This is certainly better than going about the business of design in an intuitive and haphazard manner.

High module strength and low coupling contribute to module independence by minimizing the interactions and assumptions among modules. The following three design criteria defined by Holt [7] present a good summary of these effects:

1. The complexity of a module's interactions with other modules should be less than the complexity of the module's internal structure.
2. A good module is simpler on the outside than on the inside.
3. A good module is easier to use than to build.

FURTHER GUIDELINES

In addition to strength and coupling, there are other guidelines that have an effect on module independence. These guidelines are summarized as follows.

Module Size. Module size has a bearing on a program's independence, readability, and difficulty of testing (e.g., number of paths). One could satisfy the criteria of high strength and minimal module coupling by designing a program as one huge module, but it is unlikely that high independence would be achieved by doing so. The use of a large number of modules is desirable, for modules represent explicit barriers within the program, thus reducing the potential number of interconnections among program statements and data. As a general rule, modules should contain between 10 and 100 executable high-level language statements.

Predictable Modules. A predictable module is a module whose function is independent of its past history of use. A module that internally keeps track of its own state across invocations (e.g., setting a "first-time switch") is unpredictable. All modules should be predictable, that is, they should have no "memory" from one invocation (call) to another. Interesting elusive time-dependent errors occur in programs that attempt to call an unpredictable module from several places in the program.

Decision Structure. Whenever possible it is desirable to arrange modules and decisions in those modules so that modules that are directly affected by a decision are subordinate to (called by) the module containing the decision. This tends to eliminate the passing of special parameters representing decisions to be made and also keeps decisions affecting program control at a high level in the program hierarchy.

Minimized Data Access. The amount of data that each module can reference should be minimized. Avoiding common, external, and stamp coupling is a big step in this direction. The designer should try to isolate knowledge of any particular data structure or data base record to a single module (or a small subset of modules), possibly by using informational-strength modules. The global data problem should not be solved by passing a single huge parameter list to all modules. Following these rules will minimize the scope of data access that each module has, reducing the consequences of errors and making errors easier to isolate.

Internal Procedures. An internal procedure or subroutine is a closed subroutine that physically resides in its calling module. Internal procedures should be avoided for several reasons. Internal procedures are difficult to isolate for testing (unit testing), and they cannot be called from modules other than the modules physically containing the procedures. This violates the objective of "reusability." Of course, an alternative is to insert copies of an internal procedure into all modules needing it. However, this often leads to errors (copies of the same procedure often become "nonexact copies") and complicates program maintenance (when the procedure is changed all modules using it must be recompiled). Lastly, unless a great deal of discipline is used during the coding process, internal procedures will have poor degrees of coupling with their calling modules. If a need for an internal procedure arises, the designer should consider making it a module.

COMPOSITE ANALYSIS

Module strength, coupling, and the other guidelines discussed are valuable in evaluating alternatives in a design but they do not explicitly identify the design thought process. Within composite design is a process called *composite analysis,* a top-down design reasoning process. Composite analysis involves an analysis of the problem structure and how data is transformed as it flows through the problem structure. This information is used to decompose the problem into a "layer" of modules. Each module is then viewed as a subproblem, the analysis is repeated for this subproblem, and so on.

In using composite analysis there are three basic decomposition strategies. In decomposing any subproblem, one of the following strategies is used. *STS (source/transform/sink) decomposition* involves breaking the problem into functions that acquire data, alter its form, and then deliver the data to some point outside of the problem. *Transaction decomposition* involves breaking the problem into "sister" functions that process unique types of transactions. *Functional decomposition* involves breaking the problem into functions that perform data transformations. STS decomposition is normally used to decompose the problem initially into the first layer of modules, and then either STS, transaction, or functional decomposition is used on each subproblem, the one used being dependent on characteristics of the subproblem.

Transaction and functional decomposition are basically intuitive processes and little more can be said about them. STS decomposition, however, is a more sophisticated process and can be summarized in these five steps:

1. Outline the structure of the problem, picturing it as three to ten processes based on data flow through the problem.
2. Identify the major input stream of data entering the problem and the major output data stream leaving the problem.
3. Trace the major input data stream through the problem structure. As you do this you will notice two effects: the input data stream will change form, becoming more abstract as you follow it into the problem structure, and you will eventually hit a point where the input stream seems to disappear. The point at which the input stream last appears is called the *point of highest abstraction* of the input stream.

   Perform a similar analysis of the output data stream, starting at the "end" of the problem structure and working backward. Identify the point at which the output stream first appears in its most abstract form.

These points are of interest because they divide the problem into its most independent pieces.

4. The two points identified break the problem structure into pieces (normally three). Describe these pieces of the problem structure as functions and define modules (again, normally three) that perform each of these functions. These modules become subordinate modules to the module being decomposed.
5. Define the interfaces to these modules. At this time you should be interested only in identifying the kind of data in each interface. That is, identify descriptive input and output arguments without being concerned about their precise nature (order, attributes, and representation). A subsequent design process (module external design, described in Chapter 8) will define the detail of each interface.

The decomposition process is continued down the module hierarchy, until a stopping point is reached. The general guideline telling when to stop is whenever a module is reached whose logic seems "intuitively obvious" (meaning it probably will contain 50 statements or less).

The output of the analysis process is a hierarchical block diagram showing the structural relationships of all modules (who calls whom), the functions of each module, and the interfaces among the modules. Notation for this diagram is described in Myers [2].

COMPOSITE ANALYSIS EXAMPLE

The easiest way to understand composite analysis is to see its application on an example. This same example is used in later chapters to illustrate other design and testing processes.

In deciding on an appropriate example I immediately ran into several problems. The example cannot be a large program; it must be textbook size, yet it also must not be trivial. An application program (e.g., a payroll program) might not hold the interest of system programmers and even many application programmers; a component of an operating system would have an equally limited appeal. To compromise I chose a *loader,* a program that is midway between an application program and an operating system. A second reason for selecting a loader as an example is my feeling that most readers will be at least vaguely familiar with the function of a loader.

As mentioned at the beginning of the chapter, a detailed external specification is the input to the program structure design process. Rather than supply such a specification for the loader, I simply describe the functions of the loader and its inputs and outputs in sufficient detail to allow us to design its structure.

The function of the loader is to load a program into main storage, ready for execution (some loaders also initiate the execution of the program, but this detail is disregarded here). The input to the loader is a file (referred to as INFILE) containing one or more object modules produced by a compiler. The loader can have a second input: a program library file (referred to as PROGLIB) containing a large library of object modules that may be needed in the loaded program. The loader has two outputs: the loaded program in main storage and an output file (referred to as OUTFILE) containing a memory map showing the main storage locations of the modules in the loaded program, and a list of error messages, if any.

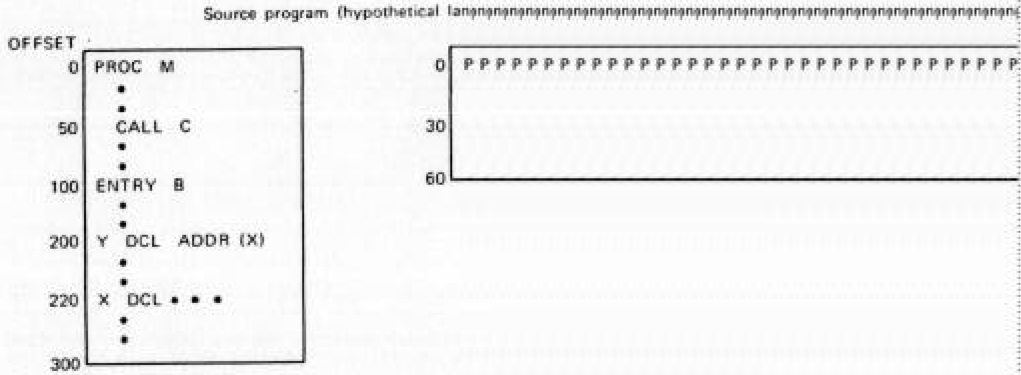
INFILE is a sequential file containing one or more object modules. PROGLIB is some type of indexed or partitioned file where object modules are stored as separate entities. All object modules have the same format: one or more ESD (external symbol dictionary) records describing external symbols and external references in the module, two or more TXT (text) records containing the object (machine) code for the module, followed by zero or more RLD (relocation dictionary) records describing any address constants in the module, and an END record. Each ESD record contains a symbol name, the type of symbol (module name, entry-point name, or reference to an external name), and the relative offset of the symbol within the module. The first TXT record contains the size of the object code for the module. Each following TXT record contains a section of the object code along with a length field describing the amount of object code on this record. The RLD records describe any address constants within the module, addresses that must be relocated when the module is assigned a particular main storage address. An RLD record contains the offset of an address constant in the object code and the number of the corresponding ESD record (each address in an object module is relative to an external symbol). For address constants that will point to areas within the module, the corresponding ESD record is the ESD for the module name (primary entry point). For address constants that will point to other modules, the corresponding ESD record is the ESD for the external reference to the other module.

To load a program, the loader has to perform the following functions:

1. Move the object code for each module in INFILE into assigned main storage locations.
2. Ensure that all external references are matched. For example, if module A is being loaded and it contains a CALL statement to module B, the loader must ensure that module B is also loaded. If module B is not in INFILE, it is loaded from PROGLIB if it is found there. Note that module B may then contain external references to other modules that must subsequently be matched.
3. Relocate all address constants. An address constant is a data area in a program that contains the address of another data area. We will assume that all address constants have a single fixed length. The compiler has no knowledge of what main storage locations the program will be loaded into, so the compiler puts *relative* addresses in the address constant fields. For an address constant pointing to a location within the same module, the compiler assigns it the offset of the location relative to the beginning of the module. Address constants pointing to external references are initialized to zero by the compiler. Once all necessary modules have been assigned main storage locations, the loader adjusts all address constants to their proper values.

Rather than explain this process in any more detail, I have described a sample input to the loader in Figure 6.1 and the resulting output in Figure 6.2. By studying Figures 6.1 and 6.2 you should be able to gather enough information to understand the design problem. In designing the loader, we assume that it executes on some operating system containing main storage allocation functions and the necessary input/output functions to perform read, write, and search operations on the input and output files.

The first step in designing the loader is to define a top module having a function equivalent to the loader. This module is called LOAD-A-PROGRAM. The next step is to view this module as a problem to be solved and use STS decomposition to break it into smaller functions. Figure 6.3 shows that this problem can be structured by data flow into five processes. The major input stream is the stream of object modules. The major output stream is the loaded program (the memory map is a secondary output). By recognizing that an object module is a module with *relative* address constants and that a loaded program is a set of modules with *absolute* address constants, the points of highest abstraction are easily found and are indicated by asterisks. The problem has now been decomposed into three functions and the module structure is started as shown in Figure 6.3.



INFILE Contents

ESD	M	MD	[illegible]
ESD	B	EP	[illegible]
ESD	C	ER	[illegible]
TXT			[illegible]
TXT	000000		
TXT	000220		
RLD	0054	3	
RLD	0200	1	
END			
ESD	C	MD	[illegible]
ESD	B	ER	[illegible]
TXT			[illegible]
TXT	000000		
RLD	0034	2	
END			

Figure 6.1 **Sample**

The remaining item to be defined
module interfaces (the ones marked 1, 2
3, we know that the output listing cons
external symbols and their absolute add
input on interface 3 is now defined to
ESTAB, a table containing external
absolute addresses. Note that we are n
data representations during the progra
ESTAB could later be defined as a seq
output on interface 3 is some yet-to-be-

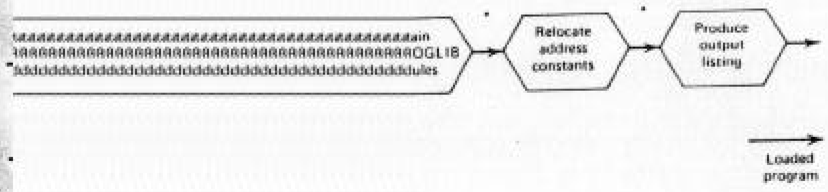
Output

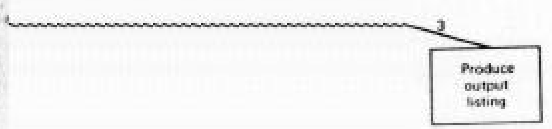
LOCATION	TYPE
100000	MOD
100100	EP
100300	MOD

age output

contain code for module M
contains 100300
contains 100220
contain code for module C
contains 100100

ponding loader output.





Interfaces

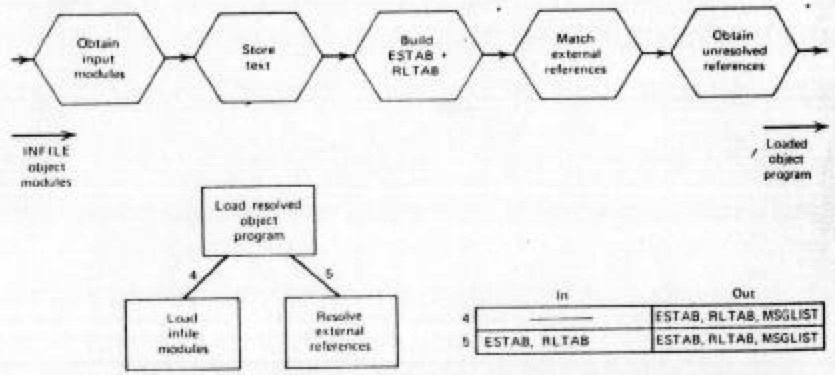
	In	Out
	———	ESTAB, RLTAB, MSGLIST
	ESTAB, RLTAB	EC
	ESTAB, MSGLIST	EC

itial decomposition.

The RELOCATE-ADCONS function needs two types of input: a description of the external symbols (particularly their absolute addresses) and a description of all address constants. Interface 2 contains two inputs: ESTAB and RLTAB, a table containing a pointer to each address constant and a pointer (or index) to the corresponding ESTAB entry. It is not necessary to pass the loaded object modules as a parameter because ESTAB points to the loaded modules. Interface 2 has one output, an error code. Interface 1 is now defined to have no inputs and three outputs: ESTAB, RLTAB, and a list of error messages for any errors encountered in the function.

The next step is to take the modules of Figure 6.3 and decompose them further. The logic of RELOCATE-ADCONS and PRODUCE-OUTPUT-LISTING is easily visualized, so they will not be further decomposed here. This leaves us with the LOAD-RESOLVED-OBJECT-PROGRAM module. The first task is to view this module as a problem to be solved and outline the problem structure as shown in Figure 6.4. The input stream is the set of object modules from INFILE and the output stream is a stored object program with all external references resolved. The input stream's point of highest abstraction occurs when all INFILE modules have been stored in memory and represented in ESTAB and RLTAB. The output stream only appears at the end-point of the problem structure. These two points break the problem into two functions, and two corresponding subordinate modules are defined.

Interface 4 has no inputs and returns three outputs: ESTAB, RLTAB, and a list of error messages. Interface 5 obviously needs ESTAB as an input. However, since the RESOLVE-EXTERNAL-REFERENCES module may have to load modules from PROGLIB, it



	In	Out
4	———	ESTAB, RLTAB, MSGLIST
5	ESTAB, RLTAB	ESTAB, RLTAB, MSGLIST

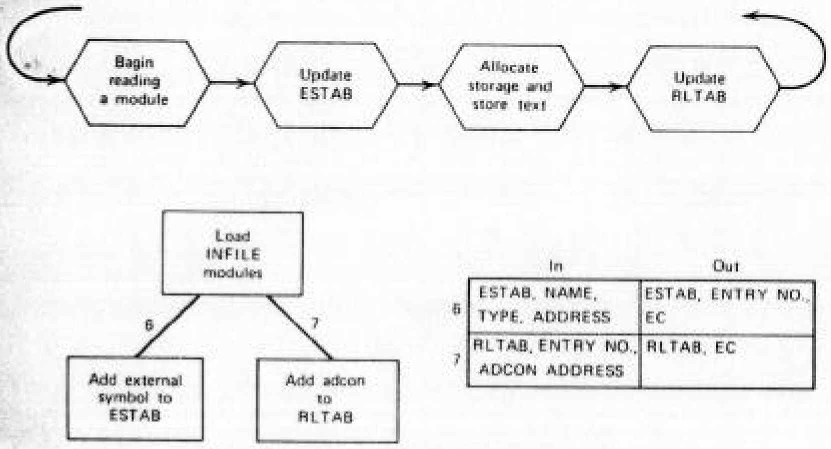
Figure 6.4 **Decomposition of LOAD-RESOLVED-OBJECT-PROGRAM.**

may have to add entries to both ESTAB and RLTAB, requiring them both as inputs and outputs.

Moving down the hierarchy, we can now attempt to decompose the LOAD-INFILE-MODULES module. Its problem structure is an iterative process as shown in Figure 6.5. The problem is simple enough so that decomposition is not necessary. However, because of a goal I have in mind (minimizing the number of modules that are aware of the representations of ESTAB and RLTAB), two functions are broken out as separate modules. Note that LOAD-INFILE-MODULES also calls the operating system for two functions: reading from INFILE and allocating blocks of main storage. ESTAB is an input on interface 6 and RLTAB is an input on interface 7, which ensures that the called modules are predictable and reentrant. One value of EC (error code) in interface 6 indicates an attempt to insert a duplicate name with type MD or EP into ESTAB.

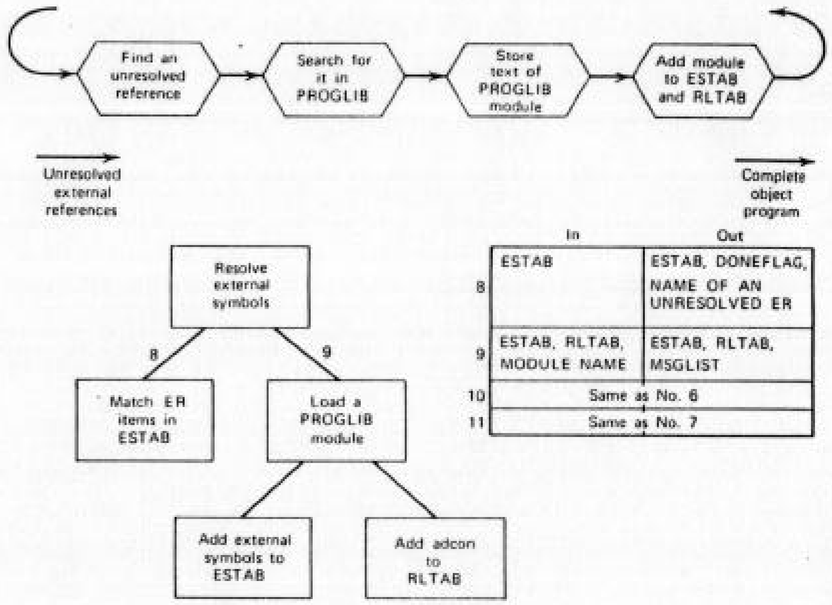
We can now search for another module to decompose; we choose RESOLVE-EXTERNAL-REFERENCES from Figure 6.4. The input stream is the set (possibly empty) of unresolved references and the output stream is a complete object program. The points of highest abstraction and the resulting decomposition are indicated in Figure 6.6.

We are faced with an interesting tradeoff at this point. Modules LOAD-PROGLIB-MODULE and LOAD-INFILE-MODULES are similar in function, suggesting the alternative of generalizing the latter module to perform both functions. This is not an unreasonable tradeoff, but I chose not to use a single module for two reasons: the files have different file organizations, and using a single module would introduce control coupling (explicitly telling the module which file to read). Note, however, that I can use the subordinate modules of LOAD-INFILE-MODULES as shown.



	In	Out
6	ESTAB, NAME, TYPE, ADDRESS	ESTAB, ENTRY NO., EC
7	RLTAB, ENTRY NO., ADCON ADDRESS	RLTAB, EC

Figure 6.5 **Decomposition of LOAD-INFILE-MODULES.**


Find an unresolved reference
Search for it in PROGLIB
Store text of PROGLIB module
Add module to ESTAB and RLTAB
Unresolved external references
Complete object program
Resolve external symbols
8
9
Match ER items in ESTAB
Load a PROGLIB module
Add external symbols to ESTAB
Add adcon to RLTAB

	In	Out
8	ESTAB	ESTAB, DONEFLAG, NAME OF AN UNRESOLVED ER
9	ESTAB, RLTAB, MODULE NAME	ESTAB, RLTAB, MSGLIST
10	Same as No. 6	
11	Same as No. 7	

Figure 6.6 **Final decomposition.**

Module MATCH-ER-ITEMS is quite simple. It scans ESTAB for unresolved names (indicated by a zero address field). When it encounters one it scans ESTAB looking for the same name of type MD or EP. If it finds one, it puts its address field into the address field for the unmatched name and then continues looking for other unmatched names. It returns when there are no symbols remaining unmatched (setting DONEFLAG) or when a name is encountered that cannot be matched (and returning the name as an output). Module RESOLVE-EXTERNAL-SYMBOLS is also quite simple; it iteratively calls MATCH-ER-ITEMS and LOAD-PROGLIB-MODULE until all symbols are matched or unmatched symbols cannot be located in PROGLIB. This is not the most efficient way to perform this function, but it is the simplest way. As Knuth [8] says, "Premature optimization is the root of all evil." After we have a working program, we can perform performance measurements and optimize the logic of one or

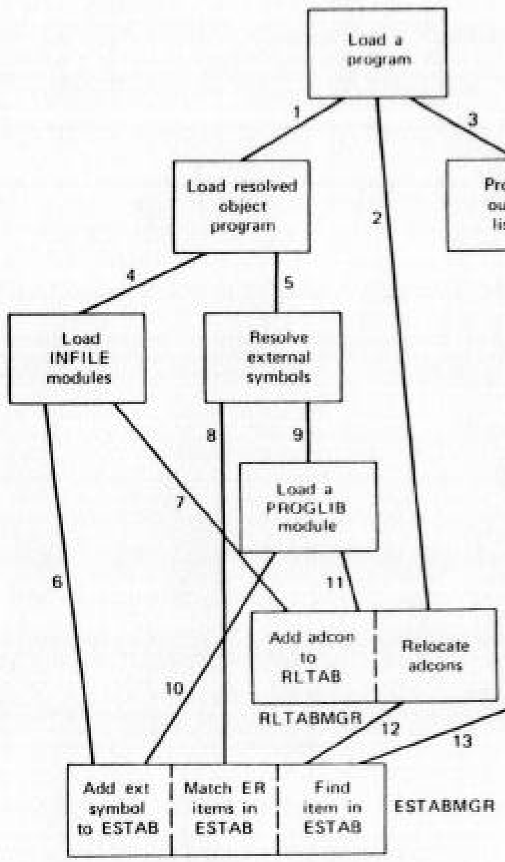


Figure 6.7 **The end result.**

Notes

1. LOAD-INFILE-MODULES uses operating system functions GET (from INFILE) and GETMAIN (to allocate storage).
2. LOAD-A-PROGLIB-MODULE uses system functions FIND and GET (from PROGLIB) and GETMAIN.
3. PRODUCE-OUTPUT-LISTING uses system function PUT (to OUTFILE).
4. ESTAB entry contains symbol name, type (MD, EP, or ER), and allocated machine address.
5. RLTAB entry contains number of corresponding ESTAB entry and the machine address of the address constant.

	In	Out
1	———	ESTAB, RLTAB, MSGLIST
2	ESTAB, RLTAB	EC
3	ESTAB, MSGLIST	EC
4	———	ESTAB, RLTAB, MSGLIST
5	ESTAB, RLTAB	ESTAB, RLTAB, MSGLIST
6	ESTAB, NAME, TYPE, ADDRESS	ESTAB, ENTRY NO., EC
7	RLTAB, ENTRY NO., ADCON ADDRESS	RLTAB, EC
8	ESTAB	ESTAB, DONE FLAG, NAME OF UNRES. ER
9	ESTAB, RLTAB, MODULE NAME	ESTAB, RLTAB, MSGLIST
10	SAME AS	NO. 6
11	SAME AS	NO. 7
12	ESTAB, ENTRY NO.	NAME, TYPE, ADDRESS, EC
13	SAME AS	NO. 12

more modules if warranted. (These points are mentioned only for the reader's understanding of the loader; we are not really interested in module logic at this time).

In scanning the current state of the design, I can find no other modules that need further decomposition. Hence we could call the design complete at this point. However, I still have the objective in mind that I mentioned earlier of minimizing the number of modules that know the attributes and representation of ESTAB and RLTAB. I can accomplish this by combining modules into informational-strength modules as shown in Figure 6.7. To isolate all knowledge of ESTAB into one module, I need to create a new function (entry point) named FIND-ITEM-IN-ESTAB, called by RELOCATE-ADCONS and PRODUCE-OUTPUT-LISTING as shown. Note that although other modules pass ESTAB and RLTAB as parameters, only the two informational-strength modules ESTABMGR and RLTABMGR are aware of the "insides" of the tables. For instance, only ESTABMGR knows the format of the ESTAB entries, whether ESTAB is a sequential table or a list, and whether ESTAB entries are sorted or unsorted.

In the final design in Figure 6.7 we have achieved several desirable results:

1. Six modules have functional strength; the other two have informational strength.
2. Each module is small and its logic is easily grasped.
3. Knowledge of ESTAB and RLTAB is hidden within single modules.
4. Only two modules are aware of the format of compiler-produced object modules.
5. Except for these two modules (which are stamp coupled), the only form of module coupling is data coupling.
6. All input/output operations to each file occur only within a single module.

Just as a reminder, an actual loader would contain one additional detail: it would either initiate execution of the loaded program or return the entry-point address of the loaded program to its caller.

VERIFICATION

Three steps can be taken to find flaws or errors in the program structure design: an n-plus-and-minus-one review, a static review, and a walk-through. The n-plus-and-minus-one review is a formal reading of the design documentation by the $n-1$ designers (authors of the system architecture and external specification), who look for translation mistakes, and by the $n+1$ designers (producers of the module external design), who check for feasibility, understandability, and compatibility with the programming language to be used and the underlying host system.

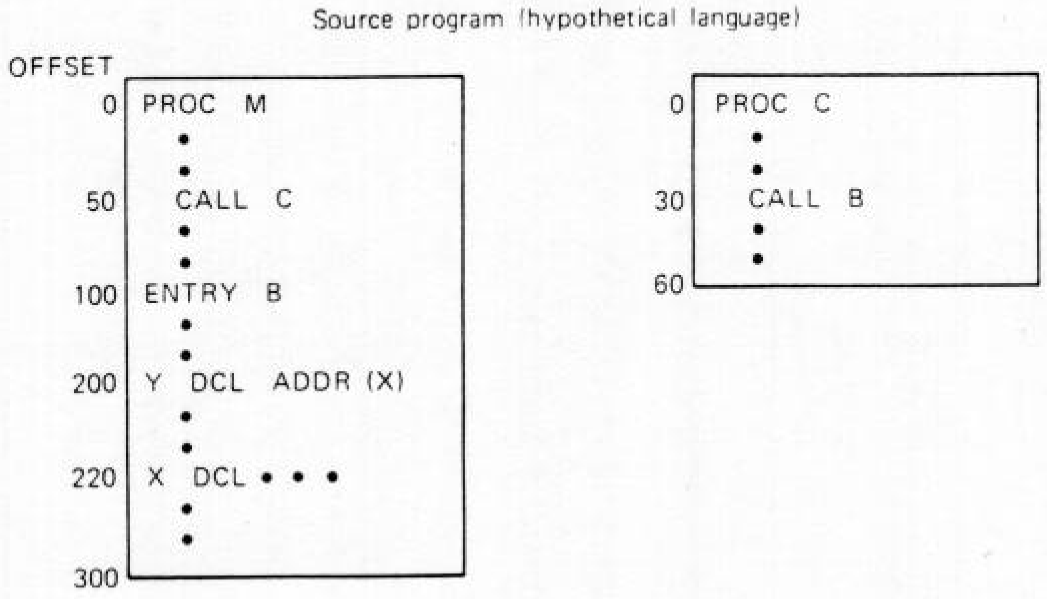
The static review is an evaluation of the design by a second party based on the guidelines discussed earlier in this chapter. The reviewer should check the design by considering such questions as: Do all modules have functional or informational strength? If not, why not? Are all modules strictly data coupled? Are all modules predictable? Is the decomposition complete (e.g., can you visualize the logic of each module)? Has the data access of each module been minimized?

The third verification step is the walk-through, similar to the walk-through methods discussed for the prior design processes. Paper test

cases are designed (e.g., Figures 6.1 and 6.2 can be used as a test case for the loader) and each test case is stepped through the module structure while keeping track of the system state. In doing this, assume that the logic of each module is correct (that each module performs its function correctly). Look for flaws in the structure such as missing functions, incomplete interfaces, and incorrect results. Use enough test cases to ensure that each module is invoked at least once. Also, include test cases for invalid inputs (e.g., an object module with no ESD records) and boundary conditions (e.g., an object module with no external references and an object module with no address constants).

REFERENCES

1. B. H. Liskov, "A Design Methodology for Reliable Software Systems," *Proceedings of the 1972 Fall Joint Computer Conference*. Montvale, N.J.: AFIPS Press, 1972, pp. 191-199.
2. G. J. Myers, *Reliable Software Through Composite Design*. New York: Petrocelli/Charter, 1975.
3. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, **15** (2), 1053-1058 (1972).
4. W. Wulf and M. Shaw, "Global Variable Considered Harmful," *SIGPLAN Notices*, **8** (2), 28-34 (1973).
5. M. J. Spier, "A Critical Look at the State of our Science," *Operating Systems Review*, **8** (2), 9-15 (1974).
6. J. B. Goodenough and D. T. Ross, "The Effect of Software Structure on Reliability, Modifiability, Reusability, Efficiency: A Preliminary Analysis," Report R-2099, SofTech Corp., Waltham, Mass., 1973.
7. R. C. Holt, "Structure of Computer Programs: A Survey," *Proceedings of the IEEE*, **63** (6), 879-893 (1975).
8. D. E. Knuth, "Structured Programming with GO TO Statements," *Computing Surveys*, **6** (4), 261-301 (1974).
9. Composite design is closely allied with the methodology called "structured design," the main differences being terminology and notation. Structured design is discussed in E. Yourdon and L. L. Constantine, *Structured Design*. New York: Yourdon, 1975.



INFILE Contents

				Notes
ESD	M	MD	0000	
ESD	B	EP	0100	
ESD	C	ER	0000	
TXT			0300	
TXT	$\boxed{000000}$			Adcon at offset 54 to point to module C
TXT	$\boxed{000220}$			Adcon at offset 200 to point to X
RLD	0054	3		
RLD	0200	1		
END				
ESD	C	MD	0000	
ESD	B	ER	0000	
TXT			0060	
TXT	$\boxed{000000}$			Adcon at offset 34 to point to entry B
RLD	0034	2		
END				

Figure 6.1 **Sample loader input.**

The remaining item to be defined at this level of the design is the module interfaces (the ones marked 1, 2, and 3). Starting with interface

OUTFILE Output

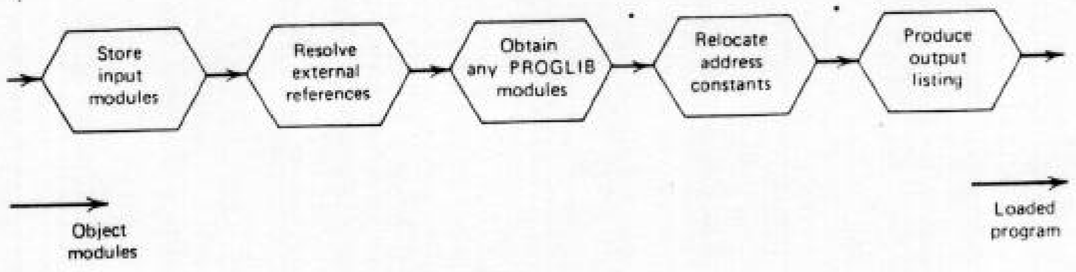
LOADER MEMORY MAP

MODULE/ENTRY PT.	LOCATION	TYPE
M	100000	MOD
B	100100	EP
C	100300	MOD

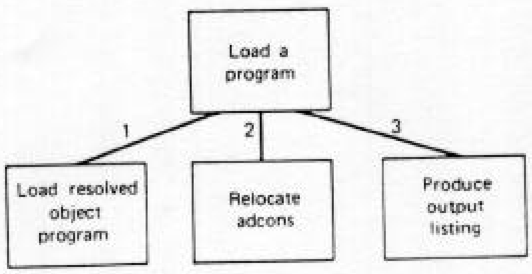
Main storage output

```
Locations 100000-1002FF contain code for module M
          location 100054 contains 100300
          location 100200 contains 100220
Locations 100300-100360 contain code for module C
          location 100334 contains 100100
```

Figure 6.2 **Corresponding loader output.**



Store input modules
Resolve external references
Obtain any PROGLIB modules
Relocate address constants
Produce output listing
Object modules
Loaded program



ESTAB:

name	type	machine address

Interfaces

	In	Out
1	———	ESTAB, RLTAB, MSGLIST
2	ESTAB, RLTAB	EC
3	ESTAB, MSGLIST	EC

RLTAB:

ESTAB entry number	machine address of adcon

Figure 6.3 **Initial decomposition.**

The RE
description
addresses)
contains tw
to each adc
ESTAB ent
parameter
has one ou
inputs and
for any erro
The nex
them furth
OUTPUT-
decompose
OBJECT-F
problem to
Figure 6.4.
and the ou
references
occurs whe
represented
the end-po
problem i
modules ar
Interfac
RLTAB, a
ESTAB as
REFEREN

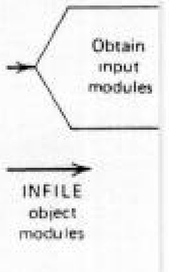
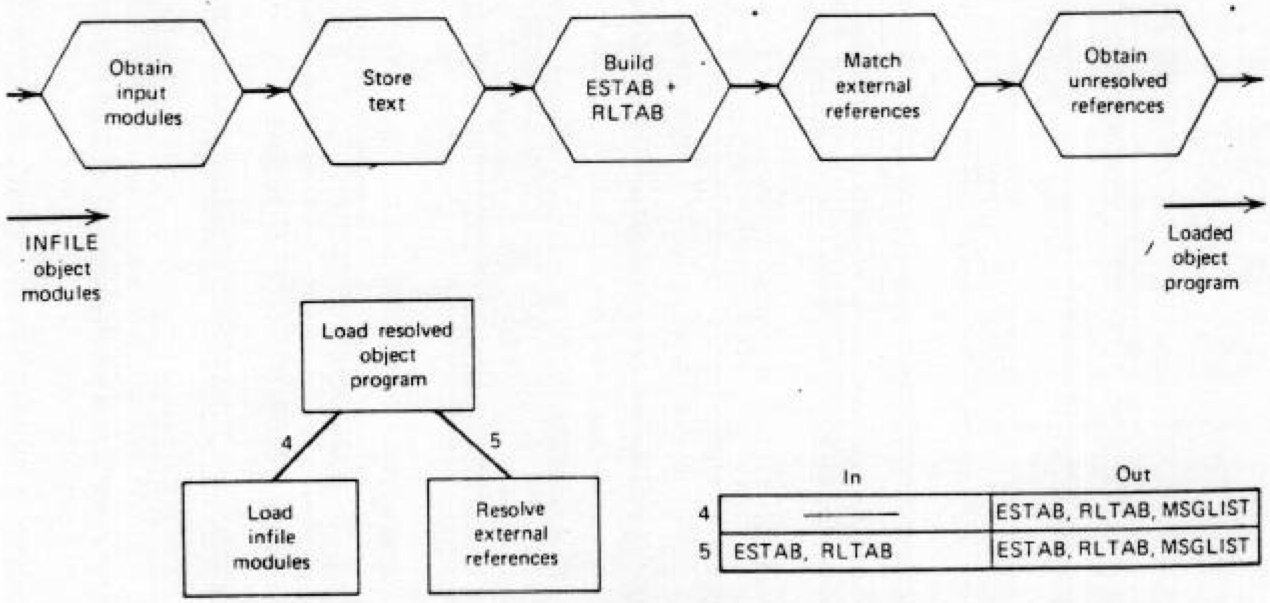


Figure 6

for any errors encountered in the function.

The next step is to take the modules of Figure 6.3 and decompose them further. The logic of RELOCATE-ADCONS and PRODUCE-OUTPUT-LISTING is easily visualized, so they will not be further decomposed here. This leaves us with the LOAD-RESOLVED-OBJECT-PROGRAM module. The first task is to view this module as a problem to be solved and outline the problem structure as shown in Figure 6.4. The input stream is the set of object modules from INFILE and the output stream is a stored object program with all external references resolved. The input stream's point of highest abstraction occurs when all INFILE modules have been stored in memory and represented in ESTAB and RLTAB. The output stream only appears at the end-point of the problem structure. These two points break the problem into two functions, and two corresponding subordinate modules are defined.

Interface 4 has no inputs and returns three outputs: ESTAB, RLTAB, and a list of error messages. Interface 5 obviously needs ESTAB as an input. However, since the RESOLVE-EXTERNAL-REFERENCES module may have to load modules from PROGLIB, it



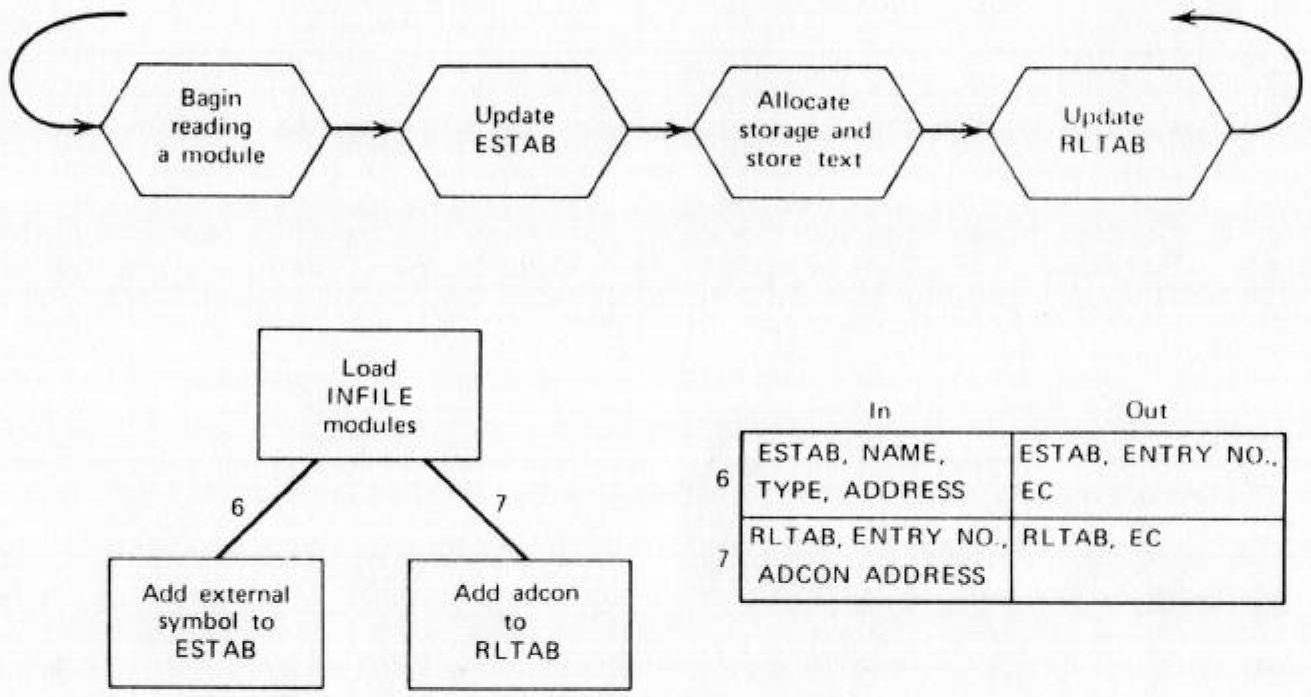
	In	Out
4	———	ESTAB, RLTAB, MSGLIST
5	ESTAB, RLTAB	ESTAB, RLTAB, MSGLIST

Figure 6.4 **Decomposition of LOAD-RESOLVED-OBJECT-PROGRAM.**

interface 6 indicates an attempt to insert a duplicate name with type MD or EP into ESTAB.

We can now search for another module to decompose; we choose RESOLVE-EXTERNAL-REFERENCES from Figure 6.4. The input stream is the set (possibly empty) of unresolved references and the output stream is a complete object program. The points of highest abstraction and the resulting decomposition are indicated in Figure 6.6.

We are faced with an interesting tradeoff at this point. Modules LOAD-PROGLIB-MODULE and LOAD-INFILE-MODULES are similar in function, suggesting the alternative of generalizing the latter module to perform both functions. This is not an unreasonable tradeoff, but I chose not to use a single module for two reasons: the files have dif-



	In	Out
6	ESTAB, NAME, TYPE, ADDRESS	ESTAB, ENTRY NO., EC
7	RLTAB, ENTRY NO., ADCON ADDRESS	RLTAB, EC

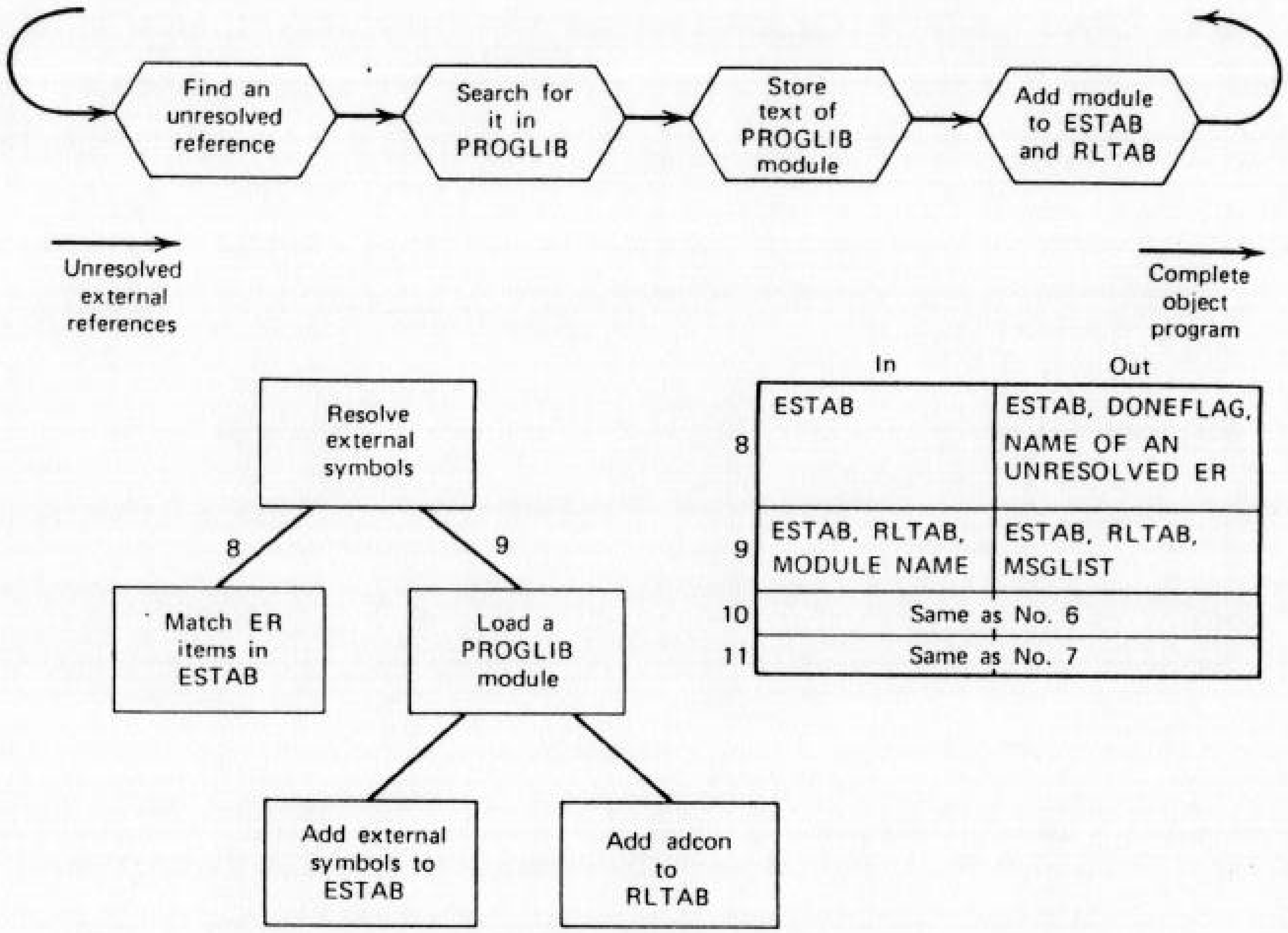
Figure 6.5 **Decomposition of LOAD-INFILE-MODULES.**

ferent
trol c
howev
MOD
M
unres
encou
or EP
the u
names
(setti
match
EXTI
MAT
symbo
PROC
but it
is the
perfor

ing them

ıpose the
an itera-
nough so
al I have
re of the
oken out
also calls
and allo-
ce 6 and
ıe called
code) in
vith type

'e choose
'he input
and the
f highest
gure 6.6.
Modules
LES are
the latter
tradeoff,
have dif-



Find an unresolved reference
Search for it in PROGLIB
Store text of PROGLIB module
Add module to ESTAB and RLTAB
Unresolved external references
Complete object program
Resolve external symbols
8
9
Match ER items in ESTAB
Load a PROGLIB module
Add external symbols to ESTAB
Add adcon to RLTAB

	In	Out
8	ESTAB	ESTAB, DONEFLAG, NAME OF AN UNRESOLVED ER
9	ESTAB, RLTAB, MODULE NAME	ESTAB, RLTAB, MSGLIST
10	Same as No. 6	
11	Same as No. 7	

Figure 6.6 **Final decomposition.**

ferent file organizations, and using a single module would introduce control coupling (explicitly telling the module which file to read). Note, however, that I can use the subordinate modules of LOAD-INFILE-MODULES as shown.

Module MATCH-ER-ITEMS is quite simple. It scans ESTAB for

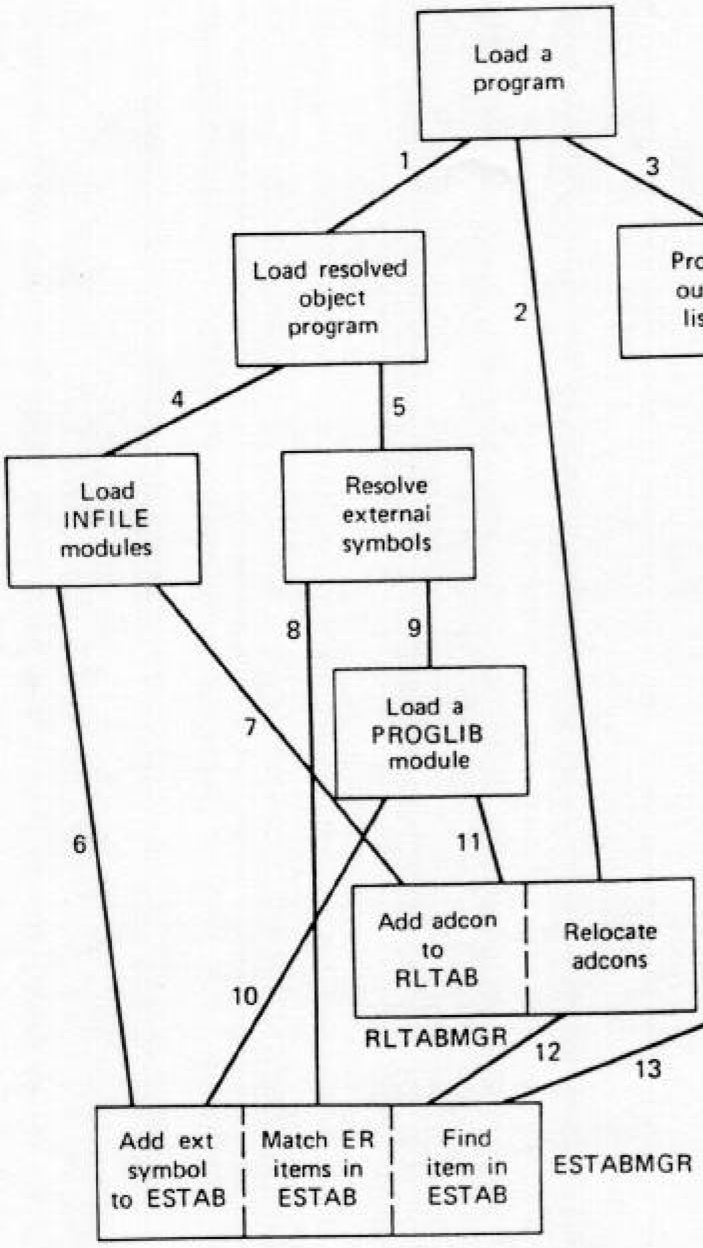
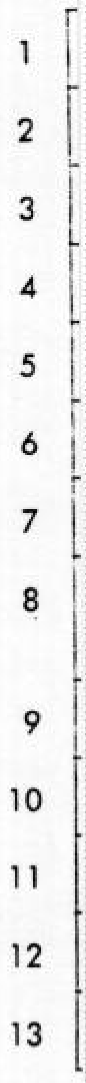


Figure 6.7 **The end result.**

Notes

1. LOAD-INFILE-MODULES uses operating system functions GET (from INFILE) and GETMAIN (to allocate storage).
2. LOAD-A-PROGLIB-MODULE uses system functions FIND and GET (from PROGLIB) and GETMAIN.
3. PRODUCE-OUTPUT-LISTING uses system function PUT (to OUTFILE).
4. ESTAB entry contains symbol name, type (MD, EP, or ER), and allocated machine address.
5. RLTAB entry contains number of corresponding ESTAB entry and the machine address of the address constant.



more
read
modu
I
modu
desig
mind
that
can,
modu
one
ITE
OUT
EST
stren