Examined are relationships between the methodology of composite design and six widely used programming languages. Strengths and weaknesses of composite design facilities of these languages are discussed. Based on this experience, language facilities for greater use of the potential of composite design are suggested.

Composite design facilities of six programming languages

by G. J. Myers

Composite design[1] is a program design technology that has the aim of structuring a program into a hierarchy of highly independent modules. Composite design has been used successfully to produce programs of high reliability and lower development, maintenance, and modification costs than might have been possible without its use. This design methodology has been used to produce applications ranging from payroll programs to report-writing programs, manufacturing-control systems to aerospace ground-support systems, and data management systems to testing tools. In reviewing such programming projects, it became apparent that certain programming languages and/or language features had eased or impeded the application of composite design. The purpose of this paper is to compare features of certain common programming languages, so that a data processing organization can improve its design standards and select appropriate programming languages for their applications.

Although the reader is assumed to have a working knowledge of composite design, a short review of the major principles is worthwhile. Composite design consists of two major sets of principles: 1. a set of principles that specify the desirable and undesirable attributes of a program structure; and 2. a set of decomposition or partitioning techniques that guide designers in a top-down definition of the hierarchical structure of programs. The first set of principles includes *module strength*, a measure of the "goodness" of a single program module, and *module coupling*, a measure of the interconnections between pairs of modules. Also, a set of additional guidelines covers such considerations as module size and predictable modules.

Modules with high strength fall into the following two categories: *functional-strength* modules and *informational-strength*

modules. A functional-strength module performs a single well-defined function. An informational-strength module is one that performs multiple functions, wherein each function is represented by a unique entry point, and performs a given transformation on the same data structure. The informational-strength module is a powerful concept because it implements the concept of *information-hiding*,[2] that is, the hiding of all knowledge of the organization of a particular data structure within a single module. The informational-strength module does not violate the single-entry-single-exit guideline of structured programming because there are no control-flow connections among the program statements associated with each entry point.

The goal of module coupling is to minimize interconnections between modules. The goal is that, where one module calls another module, all data referenced by the two modules are passed as arguments, and the arguments represent simple variables, or homogeneous lists and or arrays. Other less desirable types of module coupling, in order of decreasing desirability, are the following: two modules that reference the same nonglobal data structure (*stamp coupling*); one module that controls the logic of another (*control coupling*); two modules that reference the same global variable (*external coupling*) or global structure (*common coupling*); and one module that directly references the insides of another (*content coupling*).

Language features

To make it possible to analyze several different programming languages, a set of common language features that are related to the use of composite design are now defined. In the succeeding section, languages are compared on the basis of these features.

Since the basic construction unit of composite design is the module, the language should permit programs to be partitioned into modules. A module has the following three attributes:

- It is a closed subroutine.
- It has the potential of being independently compiled.
- It may be called by any other module in the program.

Occasionally users of composite design use *internal procedures* or subroutines as substitutes for modules. An internal procedure is a closed subroutine that is either not capable of being independently compiled or not capable of being called from every point within the program. When internal procedures are used as substitutes for modules, there are three prevalent explanations:

1. lack of a program development library to control large numbers of independent modules; 2. a feeling (sometimes valid, sometimes not) that calls to internal procedures are more efficient than calls to separate modules; and 3. the use of a language that does not provide the concept of independent modules. Although the use of internal procedures as a substitute for modules is not recommended, it is worthwhile to take account of this feature of languages.

Argument transmission mechanisms vary from language to language. The four most common mechanisms are transmission by *reference, value, name*, and *value/result*. These mechanisms are of interest because they can restrict the means by which data are transmitted between modules and—as shown in the Appendix—argument transmission mechanisms can affect the results returned by a module. That is, it is possible to construct modules that can have four different results, depending on the argument transmission mechanism used.

Since an informational strength module is a useful concept, the language should provide the concept of multiple entry point modules. Although the use of global data is contradictory to the objectives of composite design, it is worthwhile to examine the manner by which each language deals with global data. A key consideration is that a programmer should have to specifically define a variable as global. That is, the default condition should not be global.

If internal procedures are used, the question of *data scoping* arises. This raises the question of whether variables can be locally defined within the internal procedure. A related question is whether an internal procedure can share variables with its enclosing procedure, other than by receiving them as parameters (which further implies a form of external or common coupling).

The use of recursive modules is encouraged, where applicable in composite design. Thus the question of whether the language provides the concept of recursion is of interest.

When a set of modules references a data structure—which implies common or stamp coupling—a compile-time macrofacility is useful, so that the structure need only be defined once. This definition can be copied into the modules by the compiler.

Language comparison

The programming languages that are analyzed and their composite-design oriented features compared are the following: PL/I, FORTRAN IV, COBOL, APL, RPG II, and ALGOL 60. Although as-

sembler languages are also widely used, they are not considered here because assembler languages give the programmer direct access to the computer. Analysis of the assembler language features cannot be done because such analyses would depend on the structure of each individual program. The author believes that the use of assembler language should be discouraged[3] because the efficient use of higher level languages greatly reduces any advantage of assembler languages. Also the use of higher level languages facilitates the trend toward program simplicity and programmer communication.

A complete comparison of language features is not always possible, since there are some differences among compilers. Where such a difference has occurred, the language—as implemented in the IBM compilers for System/370—is used.

PL/I

PL/I is a multipurpose language, rich in its ability to express algorithms, that was developed in the middle 1960s. The design of PL/I draws constructively from the concepts of FORTRAN, COBOL, and ALGOL.

- PL/I external procedures and functions meet the definition of a module.
- PL/I internal procedures and functions meet the definition of an internal procedure.
- The default argument transmission mechanism is by reference. The default-argument transmission mechanism changes to transmission by value when the argument is enclosed in an extra set of parentheses, in a CALL statement or function reference. Default argument transmission is also by value when a number of other conditions arise, such as when the argument is a constant or an argument involves operators. Similarly, transmission is by value when the attributes of the argument conflict with those of the corresponding parameters, and the called procedure is internal.
- PL/I modules can have multiple entry points.
- The name of a variable becomes global by specifying the EXTERNAL attribute in its declaration.
- Variables can be locally declared in internal procedures. If a variable is not explicitly declared, that variable refers to the variable of the same name, which is found by searching outward through the static block structure of the external procedure.

- PL/I modules can be recursive, although they must be explicitly identified as such.

- The INCLUDE statement provides a compile-time copying facility.

- Arguments can be statement labels and entry names. Such arguments are not recommended, since they represent control coupling.

FORTRAN

FORTRAN, which is the first programming language to be called a "language," was so designated by John Backus at the 1957 Western Joint Computer Conference. The FORTRAN language is oriented toward numeric and scientific applications.

- FORTRAN subroutines and function subprograms meet the definition of a module.

- FORTRAN has no concept of internal procedures. Strictly speaking, the statement function is an internal procedure, but it can only consist of one statement.

- The default argument transmission mechanism is by value result, and the mechanism can be changed to transmission by reference by enclosing the parameter in slashes in the called module. In many FORTRAN compilers, the mechanism is always that of transmission by reference.

- FORTRAN IV modules may have multiple entry points.

- The COMMON statement defines global variables. In the blank common statement, the names of the variables are not global. The global variables are placed in a single global block of storage and are referenced by their relative position. Hence, a global variable can be referenced with different names in different modules, a fact that is frequently a source of programming errors. In a labeled common statement, variables are placed in named blocks of storage, and these names are globally known.

- Data scoping, as a language feature, does not apply to FORTRAN IV because of the absence of internal procedures.

- Recursion is similarly not permitted.

- There is no compile time copying facility.

- Arguments may be statement numbers or module names, which, again is a situation of control coupling.

COBOL

The COBOL language, which was designed in the early 1960s, is used extensively in business data processing applications.

- COBOL subprograms meet the definition of a module. The provision of COBOL subprograms, however, is a relatively recent addition to the language, and is not supported by all compilers.

- A *performed paragraph* (or section) is an internal procedure. No arguments, however, can be passed to a performed paragraph. For this reason, it is not recommended that performed paragraphs be used as substitutes for modules in composite design because module interfaces are not explicitly identified in the code (i.e., all performed paragraphs become common coupled).

- The argument transmission mechanism in CALL statements to subprograms is in the form of transmissions by reference.

- Subprograms can have multiple entry points.

- COBOL is an interesting language, in that there is no concept of global data among modules. Thus the only provision for sharing data between two modules is by passing arguments. On the other hand, if performed paragraphs are used as substitutes for modules, then all data within a program is global data.

- Variables cannot be locally declared within an internal procedure (performed paragraph). All names in the internal procedure refer to variables in the data division of the module.

- Recursion is not permitted.

- The COPY statement provides a compile-time copying facility.

APL

The APL language, which evolved during the 1960s, is oriented toward interactive terminal environments and toward vector and array processing.

- The APL *function* meets the definition of a module.

- APL has no concept of internal procedures.

- Argument transmission is by value. APL is restrictive, however, in that it permits a maximum of two input arguments to a module and one output argument. There are ways around this restriction, but none of them is desirable. A set of variables can be packaged into a vector or an array and then

passed as a single argument, but this is considered to be tricky or obscure coding, and it does not work in all cases. For example, a single argument is not sufficient when one wishes to return two results such as an array and a scalar variable that contains an error return code. The alternative is to define the variables to be global, which is also an undesirable choice. At least one implementation of APL has recognized this problem and has removed the restriction by allowing additional arguments to be transmitted by name.[4] Also, a version of APL that is termed APL.SV allows the user to simulate the effect of transmission by name by using an *execute operator*.

- APL modules cannot have multiple entry points. Hence, informational strength modules are not possible.

- In APL, any variable that is not explicitly named in the function header is defined as global. Hence, the default attribute is global. This fact contradicts the goal mentioned earlier, and is a common source of programming errors. In APL global variables are not necessarily known throughout the entire program; their scope is dynamic. When a global variable is referenced, the stack of currently suspended module activations is searched until a module is found in which the variable is declared as being local. The global variable now refers to this local variable. If the search does not encounter a module that contains this variable as a local variable, the variable becomes global to the entire program.

- APL modules can be recursive.

- There is no compile-time copying facility.

RPG II The Report Generator II (RPG II) language is oriented toward report-writing applications, which tend to be relatively small (e.g., under 100 statements). Therefore, composite design is usually of limited value to such applications. In this environment, the following points are noted.

- RPG does not provide the module concept because an RPG program consists of a single module.

- An RPG subroutine is an internal procedure.

- Argument transmission mechanisms are not available because arguments cannot be passed to subroutines.

- Subroutines cannot have multiple entry points.

- The concept of global data does not apply because of the absence of modules.
- Subroutines cannot have private locally declared data; all names in the program are known by all subroutines.
- Subroutines may not be recursive.
- A compile-time facility for defining data structures is not applicable because of the absence of modules.

ALGOL

ALGOL, which is oriented toward scientific applications, has evolved since the late 1950s. Today ALGOL and its dialects are used extensively in university environments, and are used somewhat less in industry.

- ALGOL does not provide the module concept, and, therefore, an ALGOL program cannot be composed of separately compiled modules. (A later version—ALGOL 68—removes this restriction.)
- An ALGOL subprogram is an internal procedure.
- Subroutines cannot have multiple entry points.
- The default argument transmission mechanism is that of transmission by name. The mechanism can be changed to transmission by value, by specifying the parameter as such in the procedure header.
- The concept of global data does not apply because of the absence of modules.
- Variables can be locally declared in internal procedures. If a variable is not explicitly declared, it refers to the variable of the same name found by searching outward through the static block structure of the program.
- Internal procedures can be recursive.
- There is no compile-time copying facility.

Summary and experience

A summary and comparison of language attributes is given in Table 1. Of the six languages surveyed, PL/I and FORTRAN are best suited to composite design. COBOL is also well suited, provided that modules are represented as subprograms, and not as performed paragraphs. Composite design can be used with APL,

Table 1 Summary of language attributes

	PL/I	*Fortran IV*	*Cobol*	*APL*	*RPG II*	*Algol 60*
1. Modules	Yes	Yes	Yes	Yes	No	No
2. Internal procedures	Yes	No	Yes (parameter-less)	No	Yes	Yes
3. Argument mechanism	Reference value	Value/result reference	Reference	Value	None	Name value
4. Multiple entry points	Yes	Yes	Yes	No	No	No
5. Global data	Yes	Yes	No	Yes (the default)	Not applicable	Not applicable
6. Data scoping	Static by block optional	Not applicable	Static not by block not optional	Dynamic by block optional	Static not by block not optional	Static by block optional
7. Recursion	Yes	No	No	Yes	No	Yes
8. Compile-time inclusion	Yes	No	Yes	No	No	No

although the APL problems discussed often cause undesirable compromises to be made. Composite design does not lend itself well to the design of RPG and ALGOL program: both lack the concept of a module, and RPG lacks the concept of arguments and parameters.

Features of current languages have been analyzed in light of composite design. We might also explore relationships from another view: Are there features that could further facilitate the use of composite design? Features are contemplated here that might clarify module interfaces, enhance the use of informational strength modules, and lead to automatic detection of certain classes of programming errors. Since PL/I seems to be the language that is most suitable for use with composite design, that language is used to exemplify the author's experience.

CALL and PROCEDURE statements should distinguish those arguments that are *inputs* (i.e., their values at the time of the call have some significance) and those that are *outputs* (i.e., their values may be changed by the called module).

A vital part of the composite design process is the identification of module interfaces, including an identification of objects that are inputs and those that are outputs. It would be good if this design information were carried forth into the program's source code. One way to do this is to alter the CALL statement to the following:

```
CALL MODXYZ IN (A,B,C) OUT (C,D);
```

and make a similar change to the PROCEDURE statement.

This modification has several advantages. It improves the readability of the source code because the statement conveys more information than the following conventional statement:

```
CALL MODXYZ (A,B,C,D);
```

It also gives the compiler more opportunities to detect errors. A few compilers perform a static analysis of the source code to find situations wherein a variable is referenced before it is set. Compilers cannot do this analysis on argument and parameter variables because they currently have no way of distinguishing arguments that are altered by a CALL statement and parameters that have a defined initial value. The suggested change to the CALL and PROCEDURE statements makes the distinction clear. Also, arguments in a CALL statement that appear as inputs and not outputs could thus be protected against alteration in the called module.

There should be barriers among the entry points in an informational strength module. An informational strength module can be viewed as the grouping together of all functional strength modules that have knowledge of a common data structure, thus "hiding" that data structure in a single module. The code for each entry point should be as independent as possible. Variable names and parameter names are preferably local to each entry point. Code connections among entry points (e.g., GO TOs) should be prohibited, and the only information shared by the entry points is a single declaration of the hidden data structure.

Such a barrier can be achieved in PL/L by enclosing the code for each entry point in a BEGIN block, but there are two difficulties to doing this. The BEGIN statement adds additional execution overhead, and the ENTRY statement must be placed outside the BEGIN block. This means that parameter names cannot be privately known to each entry point, whereas a new language construct might be devised to achieve these goals.

Entry points in an informational strength module should be viewed as "peers." In PL/I, however, one of these points is arbitrarily designated as the "main" entry point (the PROCEDURE name). This may seem like a trivial observation, but it does distort the intended structure of the module. One can avoid this problem by beginning each function with an ENTRY statement, and then giving the module an arbitrary procedure name (and hoping that no other modules ever call the procedure name). A better solution is to define language constructs that allow a module to have a arbitrary name (a name than cannot be called) and that allow multiple peer entry points.

Languages might be more adaptable to composite design if they were to support a new data type, called *name*. In using informa-

tional strength modules to hide data structures, one usually passes these data structures throughout the program. Only the informational strength modules, however, can know anything about the attributes and organization of fields within the structures.

The way to do this in PL/I is by using the POINTER data type. Pointer variables—because of their generality—are a frequent cause of errors.[3] The *name* data type can provide a method of passing a variable or data structure among modules without knowing anything about its attributes or organization. A module that declares a variable as a *name* should not be permitted to alter the variable nor to use it as anything but an argument or parameter.

Mechanisms might be provided to control access to global data. One problem associated with the use of global variables in large programs is that a programmer can decide to reference global variables by simply adding a declaration to a module. This makes it difficult for the project to control which modules should access which data. The situation is less severe with non-global data, since the programmer must obtain the cooperation of the programmers of other modules. The implication here is simply that programmers often take short cuts when faced with the pressures of meeting schedules or correcting errors.

Each global variable should have an *owning module* that uses a language construct to define the modules that can reference the variable and the types of references that can be made (e.g., read-only). Since the compiler cannot check the program for adherence, the linkage editor could perform the checks (i.e., when the linkage editor is about to resolve an external reference to a global variable, it determines whether the owning module has permitted such a reference).

CITED REFERENCES

1. G. J. Myers, *Reliable Software Through Composite Design*, New York: Petrocelli/Charter (1975). Also see W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal* **13**, No. 2, 115–139 (1974).
2. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM* **15**, No. 12, 1053–1058 (1972).
3. G. J. Myers, *Software Reliability: Principles and Practices*, New York: Wiley-Interscience, forthcoming.
4. R. Zaks, D. Steingart, and J. Moore, "A Firmware APL Time-Sharing System," *Proceedings of the 1971 Spring Joint Computer Conference*, Montvale, N. J. AFIPS Press, 179–190 (1971).

Appendix

An argument transmission mechanism defines the method by which data are transmitted between a calling module and the

called module. An *argument* is the name of an item of data, as it is known to the calling module, and a *parameter* is the name of an item of data, as it is known to the called module. The terminology can often be confusing because arguments are often known as *actual parameters*, and parameters are often described as *formal parameters* or *dummy arguments*. The four principal transmission mechanisms are defined as follows:

- *Transmission by reference* is a mechanism in which the *address* of the argument is transmitted to the called module. Hence, any reference in the called module to the parameter becomes a reference to the location of the argument in the calling module.

- *Transmission by value* is a mechanism in which the *value* of the argument is transmitted to the called module. In other words, the current value of the argument is assigned to the parameter. This is usually implemented by copying the value of the argument into a temporary location, and then transmitting the address of that location.

- *Transmission by name* is a mechanism in which the *name* of the argument is transmitted. This can be viewed as the textual substitution of the character string that represents the argument for all occurrences of the parameter in the called module. For examples, if the arguments are X and Y + 7 and the parameters are A and B, then all occurrences of A in the called module are replaced by the name X, and all occurrences of B are replaced by the expression Y + 7.

  Although this is the proper way to view transmission by name, it is not implemented quite as has been described. The modules are, of course, executed in their object code, not in their source code representation. In the object code representations, every reference to a parameter is compiled into a call upon a special subprogram that evaluates the address and/or value of the corresponding argument.

- *Transmission by value result* is a mechanism that is similar in implementation to transmission by value, and similar in effect to transmission by reference. When the call is executed, the value of the argument is transmitted. When the called module returns to the calling module, however, the value of the parameter is stored in the location of the argument.

To illustrate the differences among the four mechanisms, and point out the importance of understanding the mechanisms that a given language provides, the following contrived program (in a hypothetical language) produces four different results depending

Table 2 Results of four argument transmission mechanisms

Mechanisms	*Printed Values*	
	A	*B*
Reference	6	12
Value	5	0
Name	7	14
Value/Result	2	7

on which mechanism is used. In the program, variable A is a global variable.

```
GLOBAL A
B = 0
A = 1
CALL ISUB (B,A,A + 3)
PRINT A,B
END

ISUB PROC (X,Y,Z)
GLOBAL A
Y = Y + 1
A = A + Z
X = Y + A
END
```

Table 2 illustrates the four possible results.

Reprint Order No. G321-5034