Suitable modularization is a good programming practice. It permits parallel development of modules, component testing as the pieces are completed, and simpler maintenance. Achieving these properties depends on defining clean interfaces and minimizing the dependencies between modules.

Most methodologies for decomposing a program into modules place too much emphasis on control flow. Too little attention is given to how data usage should affect the choice of modules. Such conventions as COBOL's separate data division (putting all working data at the top level of the module hierarchy, and passing all information in the parameter list) give modules access to data which they do not need, but which they might use to improve local efficiency. This use of unneeded data produces dependencies among modules which logically should be unrelated.

The data which might produce unwanted dependencies is not just data unrelated to the function of the module. The primary problem occurs with data items which describe or are part of the representation of the data structures to be processed by the module. As a simple example, consider the implementation of a relational data base. The data items being related are important to the processing programs. The data items used in representing the relations—field sizes, pointers—should not be used in the processing module because that module should be insensitive to representational changes.

By choosing a suitable abstraction of the data we can present only the functionally necessary aspects of the data representation and suppress implementation details. Procedural abstraction has been used to modularize a sequence of simple actions and construct a higher level action with a simpler interface, but little use has been made of abstractions built around data. A data abstraction consists of a set of abstract objects—a data base; a print spooler; models of machines in a machine shop—and the operations which can be performed on the objects. The operations manipulate the visible aspects of the abstraction—the items in the data base, print files, the parts of the machine—and suppress access to the implementation details—the representation of the relations, the number of printers, the representation of the parts. This allows some aspects of the data abstraction to be computed rather than represented directly, and, conversely, allows aspects to be redundantly represented and insures the consistency of the representation.

Since a data abstraction includes both data objects and the operations applicable to them, a suitable modularization must include procedures for each of the operations. This multiprocedure module encapsulates the representational information and makes it inaccessible outside the module. Hence, dependencies are kept within the module as desired. Because most existing languages do not support the construction of such modules, the ideal solution would be to design new languages which do. Practical considerations, however, require that we also develop ways to use existing languages to implement multi-procedure modules. This can be done using preprocessors and management control techniques. The important thing is to recognize and consider data abstractions when the modularization is chosen, and thereby limit data access to relevant operations for manipulating it.