

## 10.1 Effective Modular Design

Modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

### 10.1.1 Functional Independence

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is measured using two qualitative criteria: cohesion and coupling.

*Cohesion* is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules.

### 10.1.2 Cohesion

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed *coincidentally cohesive*. A module that performs tasks that are related logically is *logically cohesive*. e.g., one module may read all kinds of input (from tape, disk and telecommunications port). When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion*.

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds. It performs the following tasks: (1) computes supplementary data based on original computed data, (2) produces an error report (with graphical content) on the user's workstation, (3) performs follow-up calculations requested by the user, (4) updates a database, and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists. When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present.

High cohesion is characterized by a module that performs one distinct procedural task.

### Note:

It is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

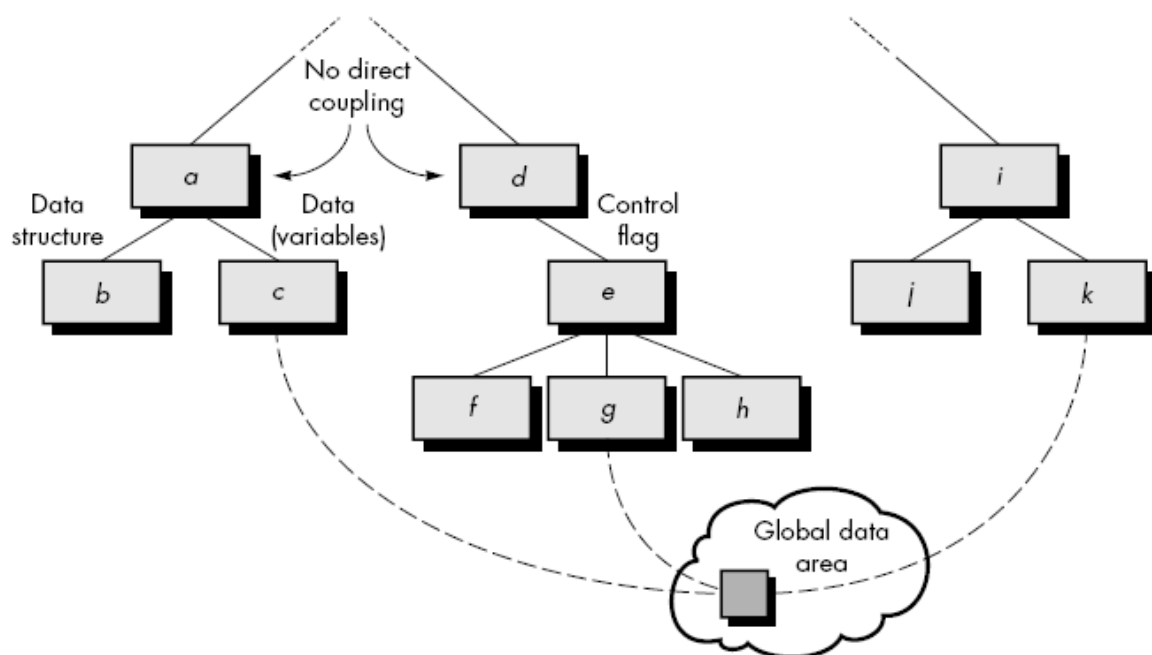
### 10.1.3 Coupling

Coupling is a measure of interconnection among modules in a software structure.

Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate through a system.

Figure 1 provides examples of different types of module coupling.



**Figure 1:** Types of coupling

Modules **a** and **d** are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module **c** is subordinate to module **a** and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called **data coupling**) is exhibited in this portion of structure. A variation of data coupling, called **stamp coupling**, is found when a

portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

At moderate levels, coupling is characterized by passage of control between modules. ***Control coupling*** is very common in most software designs and is shown in Figure 1 where a “control flag” (a variable that controls decisions in a subordinate or superordinate module) is passed between modules *d* and *e*.

High coupling occurs when a number of modules reference a global data area ***Common coupling***, as this mode is called, is shown in Figure 1. Modules *c*, *g*, and *k* each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module *c* initializes the item. Later module *g* recomputes and updates the item. Let's assume that an error occurs and *g* updates the item incorrectly. Much later in processing module, *k* reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module *k*; the actual cause, module *g*. Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, ***content coupling***, occurs when one module makes use of data or control information maintained within the boundary of another module. Secondly, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

**Notes:**

***1-Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.***

The structure shown inside the cloud in Figure 2 does not make effective use of factoring. All modules are “pancaked” below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

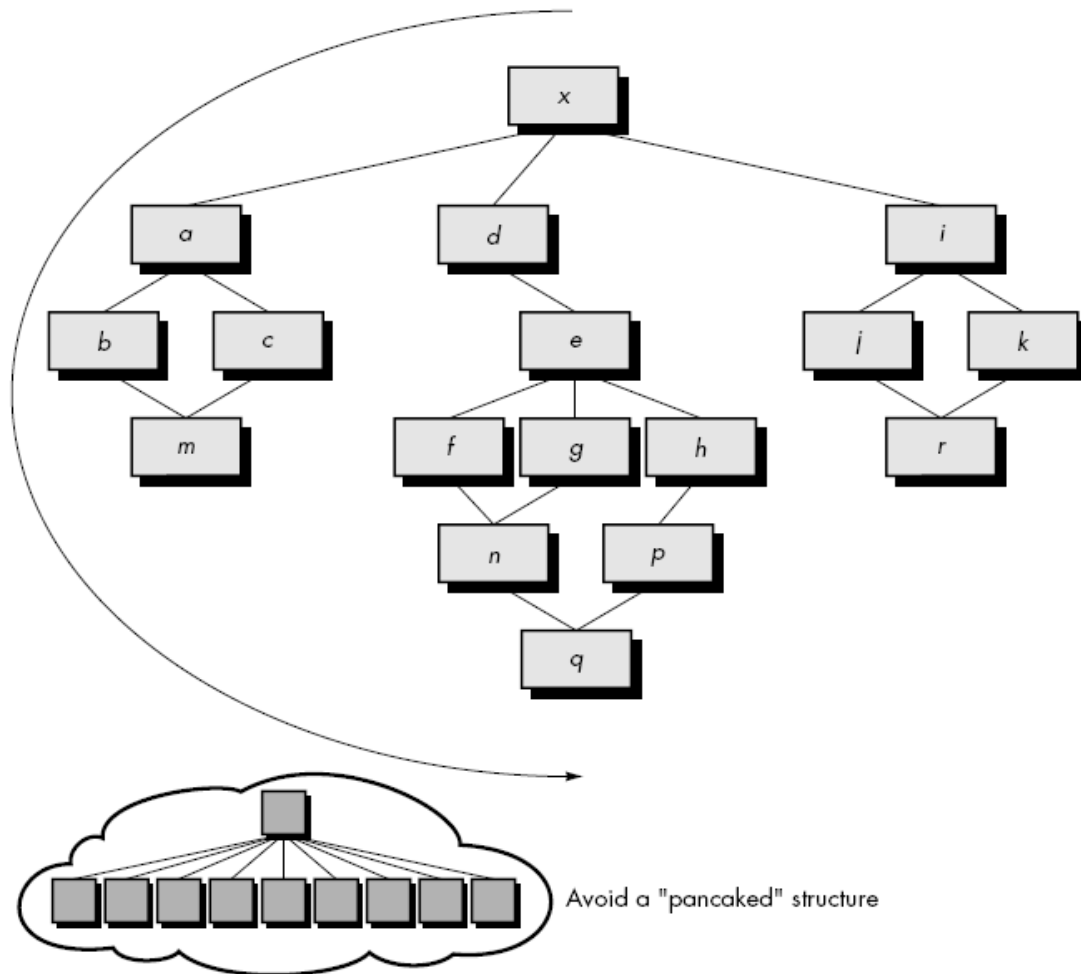


Figure 2: Program structures

***2- Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.***

Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence.

An *exploded module* becomes two or more modules in the final program structure. An *imploded module* is the result of combining the processing implied by two or more modules.

An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

