3

Modularity

From the goals of extendibility and reusability, two of the principal quality factors introduced in chapter 1, follows the need for flexible system architectures, made of autonomous software components. This is why chapter 1 also introduced the term *modularity* to cover the combination of these two quality factors.

Modular programming was once taken to mean the construction of programs as assemblies of small pieces, usually subroutines. But such a technique cannot bring real extendibility and reusability benefits unless we have a better way of guaranteeing that the resulting pieces — the **modules** — are self-contained and organized in stable architectures. Any comprehensive definition of modularity must ensure these properties.

A software construction method is modular, then, if it helps designers produce software systems made of autonomous elements connected by a coherent, simple structure. The purpose of this chapter is to refine this informal definition by exploring what precise properties such a method must possess to deserve the "modular" label. The focus will be on design methods, but the ideas also apply to earlier stages of system construction (analysis, specification) and must of course be maintained at the implementation and maintenance stages.

As it turns out, a single definition of modularity would be insufficient; as with software quality, we must look at modularity from more than one viewpoint. This chapter introduces a set of complementary properties: five *criteria*, five *rules* and five *principles* of modularity which, taken collectively, cover the most important requirements on a modular design method.

For the practicing software developer, the principles and the rules are just as important as the criteria. The difference is simply one of causality: the criteria are mutually independent — and it is indeed possible for a method to satisfy one of them while violating some of the others — whereas the rules follow from the criteria and the principles follow from the rules.

You might expect this chapter to begin with a precise description of what a module looks like. This is not the case, and for a good reason: our goal for the exploration of modularity issues, in this chapter and the next two, is precisely to analyze the properties which a satisfactory module structure must satisfy; so the form of modules will be a conclusion of the discussion, not a premise. Until we reach that conclusion the word

"module" will denote the basic unit of decomposition of our systems, whatever it actually is. If you are familiar with non-object-oriented methods you will probably think of the subroutines present in most programming and design languages, or perhaps of packages as present in Ada and (under a different name) in Modula. The discussion will lead in a later chapter to the O-O form of module — the class — which supersedes these ideas. If you have encountered classes and O-O techniques before, you should still read this chapter to understand the requirements that classes address, a prerequisite if you want to use them well.

3.1 FIVE CRITERIA

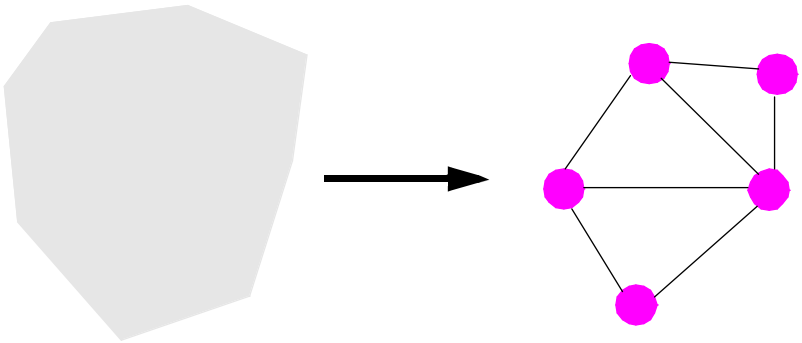
A design method worthy of being called "modular" should satisfy five fundamental requirements, explored in the next few sections:

- Decomposability.
- Composability.
- Understandability.
- Continuity.
- Protection.

Modular decomposability

> A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them

The process will often be self-repeating since each subproblem may still be complex enough to require further decomposition.



Decomposability

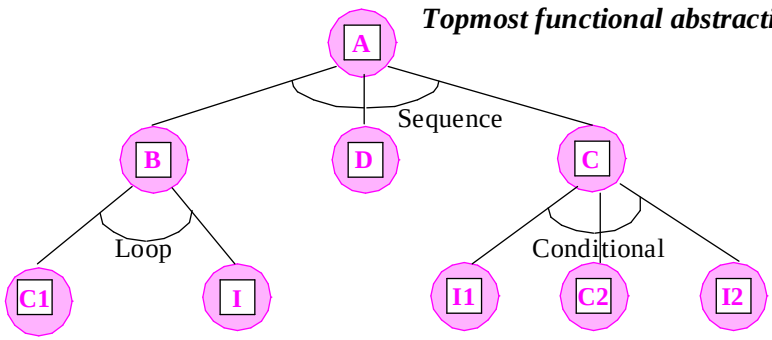
A corollary of the decomposability requirement is *division of labor*: once you have decomposed a system into subsystems you should be able to distribute work on these subsystems among different people or groups. This is a difficult goal since it limits the dependencies that may exist between the subsystems:

- You must keep such dependencies to the bare minimum; otherwise the development of each subsystem would be limited by the pace of the work on the other subsystems.
- The dependencies must be known: if you fail to list all the relations between subsystems, you may at the end of the project get a set of software elements that appear to work individually but cannot be put together to produce a complete system satisfying the overall requirements of the original problem.

As discussed below, top-down design is not as well suited to other modularity criteria.

The most obvious *example* of a method meant to satisfy the decomposability criterion is **top-down design**. This method directs designers to start with a most abstract description of the system's function, and then to refine this view through successive steps, decomposing each subsystem at each step into a small number of simpler subsystems, until all the remaining elements are of a sufficiently low level of abstraction to allow direct implementation. The process may be modeled as a tree.

A top-down hierarchy



The term "temporal cohesion" comes from the method known as structured design; see the bibliographical notes.

A typical *counter-example* is any method encouraging you to include, in each software system that you produce, a global initialization module. Many modules in a system will need some kind of initialization — actions such as the opening of certain files or the initialization of certain variables, which the module must execute before it performs its first directly useful tasks. It may seem a good idea to concentrate all such actions, for all modules of the system, in a module that initializes everything for everybody. Such a module will exhibit good "temporal cohesion" in that all its actions are executed at the same stage of the system's execution. But to obtain this temporal cohesion the method would endanger the autonomy of modules: you will have to grant the initialization module authorization to access many separate data structures, belonging to the various modules of the system and requiring specific initialization actions. This means that the author of the initialization module will constantly have to peek into the internal data structures of the other modules, and interact with their authors. This is incompatible with the decomposability criterion.

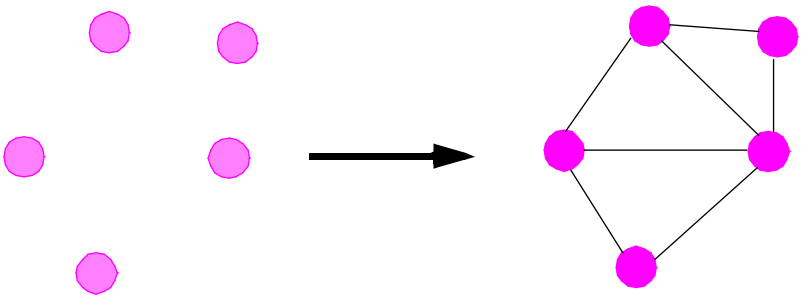
> In the object-oriented method, every module will be responsible for the initialization of its own data structures.

Modular composability

> A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

Where decomposability was concerned with the derivation of subsystems from overall systems, composability addresses the reverse process: extracting existing software elements from the context for which they were originally designed, so as to use them again in different contexts.

Composability



A modular design method should facilitate this process by yielding software elements that will be sufficiently autonomous — sufficiently independent from the immediate goal that led to their existence — as to make the extraction possible.

Composability is directly connected with the goal of reusability: the aim is to find ways to design software elements performing well-defined tasks and usable in widely different contexts. This criterion reflects an old dream: transforming the software design process into a construction box activity, so that we would build programs by combining standard prefabricated elements.

- *Example 1*: *subprogram libraries*. Subprogram libraries are designed as sets of composable elements. One of the areas where they have been successful is numerical computation, which commonly relies on carefully designed subroutine libraries to solve problems of linear algebra, finite elements, differential equations etc.
- *Example 2*: *Unix Shell conventions*. Basic Unix commands operate on an input viewed as a sequential character stream, and produce an output with the same standard structure. This makes them potentially composable through the | operator of the command language ("shell"): *A* | *B* represents a program which will take *A*'s input, have *A* process it, send the output to *B* as input, and have it processed by *B*. This systematic convention favors the composability of software tools.
- *Counter-example*: *preprocessors*. A popular way to extend the facilities of programming languages, and sometimes to correct some of their deficiencies, is to

use "preprocessors" that accept an extended syntax as input and map it into the standard form of the language. Typical preprocessors for Fortran and C support graphical primitives, extended control structures or database operations. Usually, however, such extensions are not compatible; then you cannot combine two of the preprocessors, leading to such dilemmas as whether to use graphics or databases.

The figure illustrating top-down design was on page 41.

Composability is independent of decomposability. In fact, these criteria are often at odds. Top-down design, for example, which we saw as a technique favoring decomposability, tends to produce modules that are *not* easy to combine with modules coming from other sources. This is because the method suggests developing each module to fulfill a specific requirement, corresponding to a subproblem obtained at some point in the refinement process. Such modules tend to be closely linked to the immediate context that led to their development, and unfit for adaptation to other contexts. The method provides neither hints towards making modules more general than immediately required, nor any incentives to do so; it helps neither avoid nor even just detect commonalities or redundancies between modules obtained in different parts of the hierarchy.

That composability and decomposability are both part of the requirements for a modular method reflects the inevitable mix of top-down and bottom-up reasoning — a complementarity that René Descartes had already noted almost four centuries ago, as shown by the contrasting two paragraphs of the *Discourse* extract at the beginning of part B.

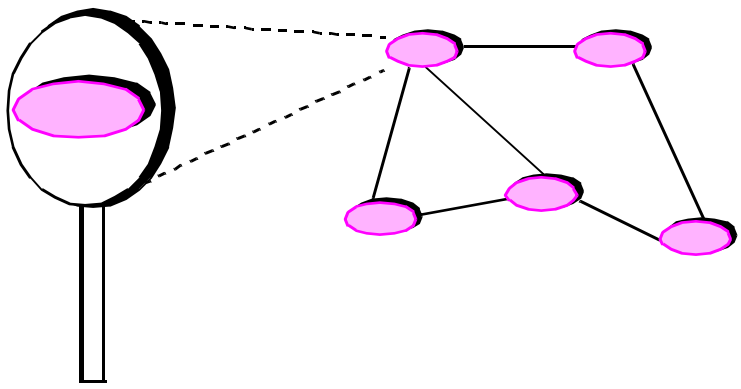
Modular understandability

> A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

See "ABOUT SOFTWARE MAINTENANCE", 1.3, page 17.

The importance of this criterion follows from its influence on the maintenance process. Most maintenance activities, whether of the noble or not-so-noble category, involve having to dig into existing software elements. A method can hardly be called modular if a reader of the software is unable to understand its elements separately.

Understandability



This criterion, like the others, applies to the modules of a system description at any level: analysis, design, implementation.

- *Counter-example*: *sequential dependencies*. Assume some modules have been so designed that they will only function correctly if activated in a certain prescribed order; for example, *B* can only work properly if you execute it after *A* and before *C*, perhaps because they are meant for use in "piped" form as in the Unix notation encountered earlier:

  A | *B* | *C*

  Then it is probably hard to understand *B* without understanding *A* and *C* too.

In later chapters, the modular understandability criterion will help us address two important questions: how to document reusable components; and how to index reusable components so that software developers can retrieve them conveniently through queries. The criterion suggests that information about a component, useful for documentation or for retrieval, should whenever possible appear in the text of the component itself; tools for documentation, indexing or retrieval can then process the component to extract the needed pieces of information. Having the information included *in* each component is preferable to storing it elsewhere, for example in a database of information *about* components.

See also, later in this chapter, "Self-Documentation", page 54.

Modular continuity

> A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.

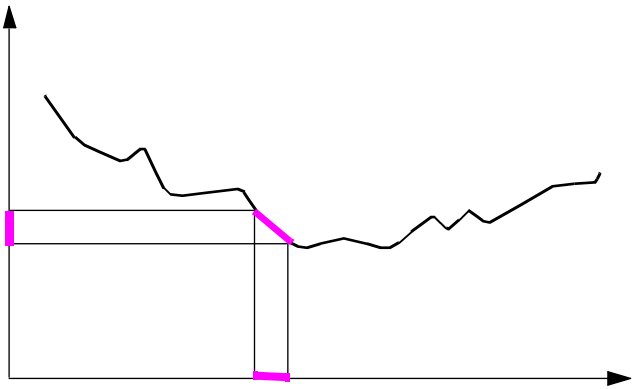
This criterion is directly connected to the general goal of extendibility. As emphasized in an earlier chapter, change is an integral part of the software construction process. The requirements will almost inevitably change as the project progresses. Continuity means that small changes should affect individual modules in the structure of the system, rather than the structure itself.

See "Extendibility", page 6.

The term "continuity" is drawn from an analogy with the notion of a continuous function in mathematical analysis. A mathematical function is continuous if (informally) a small change in the argument will yield a proportionally small change in the result. Here the function considered is the software construction method, which you can view as a mechanism for obtaining systems from specifications:

software_construction_method: *Specification* → *System*

Continuity



This mathematical term only provides an analogy, since we lack formal notions of size for software. More precisely, it would be possible to define a generally acceptable measure of what constitutes a "small" or "large" change to a program; but doing the same for the specifications is more of a challenge. If we make no pretense of full rigor, however, the concepts should be intuitively clear and correspond to an essential requirement on any modular method.

This will be one of our principles of style: *Symbolic Constant Principle, page 884.*

- *Example 1*: *symbolic constants*. A sound style rule bars the instructions of a program from using any numerical or textual constant directly; instead, they rely on symbolic names, and the actual values only appear in a constant definition (**constant** in Pascal or Ada, preprocessor macros in C, *PARAMETER* in Fortran 77, constant attributes in the notation of this book). If the value changes, the only thing to update is the constant definition. This small but important rule is a wise precaution for continuity since constants, in spite of their name, are remarkably prone to change.

See "Uniform Access", page 55.

- *Example 2*: *the Uniform Access principle*. Another rule states that a single notation should be available to obtain the features of an object, whether they are represented as data fields or computed on demand. This property is sufficiently important to warrant a separate discussion later in this chapter.
- *Counter-example 1*: *using physical representations*. A method in which program designs are patterned after the physical implementation of data will yield designs that are very sensitive to slight changes in the environment.
- *Counter-example 2*: *static arrays*. Languages such as Fortran or standard Pascal, which do not allow the declaration of arrays whose bounds will only be known at run time, make program evolution much harder.

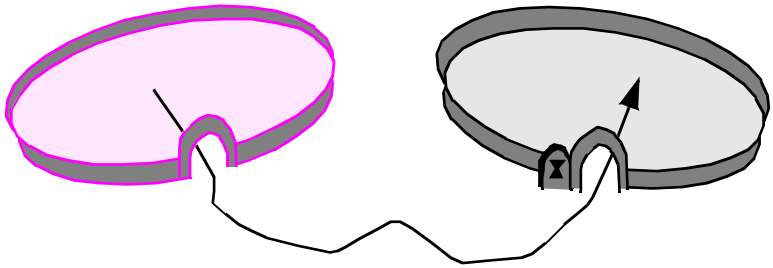
Modular protection

> A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

The underlying issue, that of failures and errors, is central to software engineering. The errors considered here are run-time errors, resulting from hardware failures, erroneous input or exhaustion of needed resources (for example memory storage). The criterion does not address the avoidance or correction of errors, but the aspect that is directly relevant to modularity: their propagation.

The question of how to handle abnormal cases is discussed in detail in chapter 12.

Protection violation



- *Example*: *validating input at the source*. A method requiring that you make every module that inputs data also responsible for checking their validity is good for modular protection.
- *Counter-example*: *undisciplined exceptions*. Languages such as PL/I, CLU, Ada, C++ and Java support the notion of exception. An exception is a special signal that may be "raised" by a certain instruction and "handled" in another, possibly remote part of the system. When the exception is raised, control is transferred to the handler. (Details of the mechanism vary between languages; Ada or CLU are more disciplined in this respect than PL/I.) Such facilities make it possible to decouple the algorithms for normal cases from the processing of erroneous cases. But they must be used carefully to avoid hindering modular protection. The chapter on exceptions will investigate how to design a disciplined exception mechanism satisfying the criterion.

More on this topic in "Assertions are not an input checking mechanism", page 346

On exception handling, see chapter 12.

3.2 FIVE RULES

From the preceding criteria, five rules follow which we must observe to ensure modularity:

- Direct Mapping.
- Few Interfaces.
- Small interfaces (weak coupling).
- Explicit Interfaces.
- Information Hiding.

The first rule addresses the connection between a software system and the external systems with which it is connected; the next four all address a common issue — how modules will communicate. Obtaining good modular architectures requires that communication occur in a controlled and disciplined way.

Direct Mapping

Any software system attempts to address the needs of some problem domain. If you have a good model for describing that domain, you will find it desirable to keep a clear correspondence (mapping) between the structure of the solution, as provided by the software, and the structure of the problem, as described by the model. Hence the first rule:

> The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.

This advice follows in particular from two of the modularity criteria:

- Continuity: keeping a trace of the problem's modular structure in the solution's structure will make it easier to assess and limit the impact of changes.
- Decomposability: if some work has already been done to analyze the modular structure of the problem domain, it may provide a good starting point for the modular decomposition of the software.

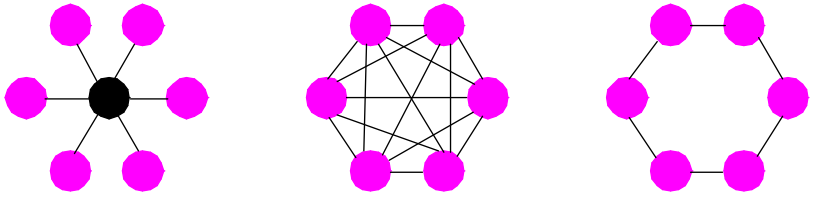
Few Interfaces

The Few Interfaces rule restricts the overall number of communication channels between modules in a software architecture:

> Every module should communicate with as few others as possible.

Communication may occur between modules in a variety of ways. Modules may call each other (if they are procedures), share data structures etc. The Few Interfaces rule limits the number of such connections.

Types of module interconnection structures



(A) **(B)** **(C)**

More precisely, if a system is composed of n modules, then the number of intermodule connections should remain much closer to the minimum, $n–1$, shown as **(A)** in the figure, than to the maximum, $n\ (n-1)\ /2$, shown as **(B)**.

This rule follows in particular from the criteria of continuity and protection: if there are too many relations between modules, then the effect of a change or of an error may

propagate to a large number of modules. It is also connected to composability (if you want a module to be usable by itself in a new environment, then it should not depend on too many others), understandability and decomposability.

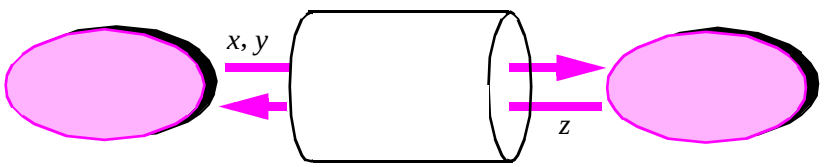
Case **(A)** on the last figure shows a way to reach the minimum number of links, $n - 1$, through an extremely centralized structure: one master module; everybody else talks to it and to it only. But there are also much more "egalitarian" structures, such as **(C)** which has almost the same number of links. In this scheme, every module just talks to its two immediate neighbors, but there is no central authority. Such a style of design is a little surprising at first since it does not conform to the traditional model of functional, top-down design. But it can yield robust, extendible architectures; this is the kind of structure that object-oriented techniques, properly applied, will tend to yield.

Small Interfaces

The Small Interfaces or "Weak Coupling" rule relates to the size of intermodule connections rather than to their number:

> If two modules communicate, they should exchange as little information as possible

An electrical engineer would say that the channels of communication between modules must be of limited bandwidth:



Communication bandwidth between modules

The Small Interfaces requirement follows in particular from the criteria of continuity and protection.

An extreme *counter-example* is a Fortran practice which some readers will recognize: the "garbage common block". A common block in Fortran is a directive of the form

COMMON /common_name/ variable$_1$*,… variable*$_n$

indicating that the variables listed are accessible not just to the enclosing module but also to any other module which includes a *COMMON* directive with the same *common_name*. It is not infrequent to see Fortran systems whose every module includes an identical gigantic *COMMON* directive, listing all significant variables and arrays so that every module may directly use every piece of data.

The problem, of course, is that every module may also misuse the common data, and hence that modules are tightly coupled to each other; the problems of modular continuity (propagation of changes) and protection (propagation of errors) are particularly nasty. This time-honored technique has nevertheless remained a favorite, no doubt accounting for many a late-night debugging session.

Developers using languages with nested structures can suffer from similar troubles. With block structure as introduced by Algol and retained in a more restricted form by Pascal, it is possible to include blocks, delimited by **begin** … **end** pairs, within other blocks. In addition every block may introduce its own variables, which are only meaningful within the syntactic scope of the block. For example:

The Body of a block is a sequence of instructions. The syntax used here is compatible with the notation used in subsequent chapters, so it is not exactly Algol's. "--" introduces a comment.

```
local-- Beginning of block B1
        x, y: INTEGER
do
        … Instructions of B1 …
        local -- Beginning of block B2
                z: BOOLEAN
        do
                … Instructions of B2 …
        end --- of block B2

        local -- Beginning of block B3
                y, z: INTEGER
        do
                … Instructions of B3 …
        end -- of block B3

        … Instructions of B1 (continued) …
end -- of block B1
```

Variable *x* is accessible to all instructions throughout this extract, whereas the two variables called *z* (one *BOOLEAN*, the other *INTEGER*) have scopes limited to B2 and B3 respectively. Like *x*, variable *y* is declared at the level of B1, but its scope does not include B3, where another variable of the same name (and also of type *INTEGER*) locally takes precedence over the outermost *y*. In Pascal this form of block structure exists only for blocks associated with routines (procedures and functions).

With block structure, the equivalent of the Fortran garbage common block is the practice of declaring all variables at the topmost level. (The equivalent in C-based languages is to introduce all variables as external.)

On clusters see chapter 28. The O-O alternative to nesting is studied in "The architectural role of selective exports", page 209.

Block structure, although an ingenious idea, introduces many opportunities to violate the Small Interfaces rule. For that reason we will refrain from using it in the object-oriented notation devised later in this book, especially since the experience of Simula, an object-oriented Algol derivative supporting block structure, shows that the ability to nest classes is redundant with some of the facilities provided by inheritance. The architecture

of object-oriented software will involve three levels: a system is a set of clusters; a cluster is a set of classes; a class is a set of features (attributes and routines). Clusters, an organizational tool rather than a linguistic construct, can be nested to allow a project leader to structure a large system in as many levels as necessary; but classes as well as features have a flat structure, since nesting at either of those levels would cause unnecessary complication.

Explicit Interfaces

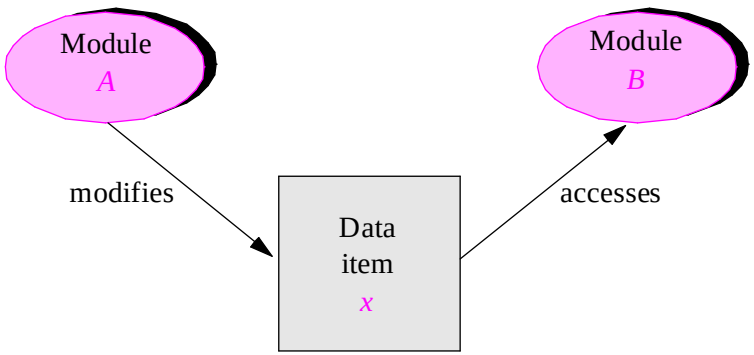
With the fourth rule, we go one step further in enforcing a totalitarian regime upon the society of modules: not only do we demand that any conversation be limited to few participants and consist of just a few words; we also require that such conversations must be held in public and loudly!

> Whenever two modules *A* and *B* communicate, this must be obvious from the text of *A* or *B* or both.

Behind this rule stand the criteria of decomposability and composability (if you need to decompose a module into several submodules or compose it with other modules, any outside connection should be clearly visible), continuity (it should be easy to find out what elements a potential change may affect) and understandability (how can you understand *A* by itself if *B* can influence its behavior in some devious way?).

One of the problems in applying the Explicit Interfaces rule is that there is more to intermodule coupling than procedure call; data sharing, in particular, is a source of indirect coupling:

Data sharing



Assume that module *A* modifies and module *B* uses the same data item *x*. Then *A* and *B* are in fact strongly coupled through *x* even though there may be no apparent connection, such as a procedure call, between them.

Information Hiding

The rule of Information Hiding may be stated as follows:

> The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

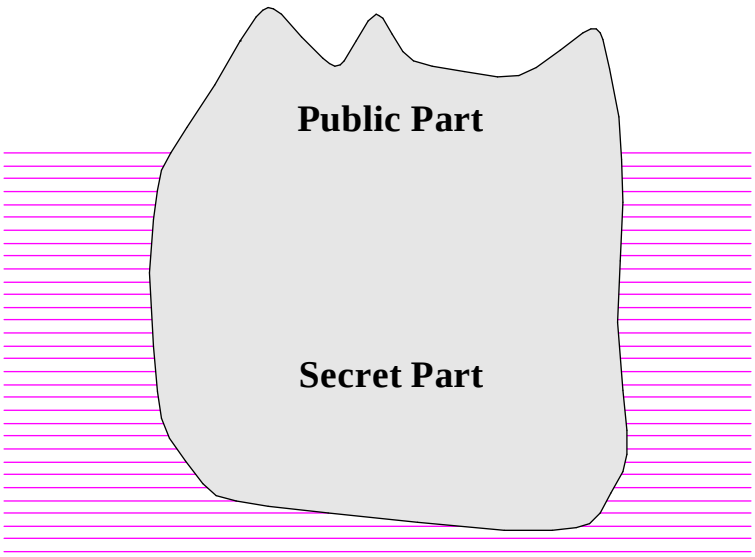
Application of this rule assumes that every module is known to the rest of the world (that is to say, to designers of other modules) through some official description, or **public** properties.

Of course, the whole text of the module itself (program text, design text) could serve as the description: it provides a correct view of the module since it *is* the module! The Information Hiding rule states that this should not in general be the case: the description should only include *some* of the module's properties. The rest should remain non-public, or **secret**. Instead of public and secret properties, one may also talk of exported and private properties. The public properties of a module are also known as the **interface** of the module (not to be confused with the user interface of a software system).

The fundamental reason behind the rule of Information Hiding is the continuity criterion. Assume a module changes, but the changes apply only to its secret elements, leaving the public ones untouched; then other modules who use it, called its *clients*, will not be affected. The smaller the public part, the higher the chances that changes to the module will indeed be in the secret part.

We may picture a module supporting Information Hiding as an iceberg; only the tip — the interface — is visible to the clients.

A module under Information Hiding



As a typical example, consider a procedure for retrieving the attributes associated with a key in a certain table, such as a personnel file or the symbol table of a compiler. The procedure will internally be very different depending on how the table is stored (sequential array or file, hash table, binary or B-Tree etc.). Information hiding implies that uses of this procedure should be independent of the particular implementation chosen. That way client modules will not suffer from any change in implementation.

Information hiding emphasizes separation of function from implementation. Besides continuity, this rule is also related to the criteria of decomposability, composability and understandability. You cannot develop the modules of a system separately, combine various existing modules, or understand individual modules, unless you know precisely what each of them may and may not expect from the others.

Which properties of a module should be public, and which ones secret? As a general guideline, the public part should include the specification of the module's functionality; anything that relates to the implementation of that functionality should be kept secret, so as to preserve other modules from later reversals of implementation decisions.

This first answer is still fairly vague, however, as it does not tell us what is the specification and what is the implementation; in fact, one might be tempted to reverse the definition by stating that the specification consists of whatever public properties the module has, and the implementation of its secrets! The object-oriented approach will give us a much more precise guideline thanks to the theory of abstract data types.

See chapter 6, in particular "Abstract data types and information hiding", page 144.

To understand information hiding and apply the rule properly, it is important to avoid a common misunderstanding. In spite of its name, information hiding does not imply *protection* in the sense of security restrictions — physically prohibiting authors of client modules from accessing the internal text of a supplier module. Client authors may well be permitted to read all the details they want: preventing them from doing so may be reasonable in some circumstances, but it is a project management decision which does not necessarily follow from the information hiding rule. As a technical requirement, information hiding means that client modules (whether or not their authors are permitted to read the secret properties of suppliers) should only rely on the suppliers' public properties. More precisely, it should be impossible to write client modules whose correct functioning depends on secret information.

> In a completely formal approach to software construction, this definition would be stated as follows. To prove the correctness of a module, you will need to assume some properties about its suppliers. Information hiding means that such proofs are only permitted to rely on public properties of the suppliers, never on their secret properties.

See the comments on conditional correctness on page 4.

Consider again the example of a module providing a table searching mechanism. Some client module, which might belong to a spreadsheet program, uses a table, and relies on the table module to look for a certain element in the table. Assume further that the algorithm uses a binary search tree implementation, but that this property is secret — not part of the interface. Then you may or may not allow the author of the table searching module to tell the author of the spreadsheet program what implementation he has used for tables. This is a project management decision, or perhaps (for commercially released software) a marketing decision; in either case it is irrelevant to the question of information

hiding. Information hiding means something else: that *even if the author of the spreadsheet program knows* that the implementation uses a binary search tree, he should be unable to write a client module which will only function correctly with this implementation — and would not work any more if the table implementation was changed to something else, such as hash coding.

One of the reasons for the misunderstanding mentioned above is the very term "information hiding", which tends to suggest physical protection. "Encapsulation", sometimes used as a synonym for information hiding, is probably preferable in this respect, although this discussion will retain the more common term.

By default "Ada" always means the most widespread form of the language (83), not the more recent Ada 95. Chapter 33 presents both versions.

As a summary of this discussion: the key to information hiding is not management or marketing policies as to who may or may not access the source text of a module, but strict **language rules** to define what access rights a module has on properties of its suppliers. As explained in the next chapter, "encapsulation languages" such as Ada and Modula-2 made the first steps in the right direction. Object technology will bring a more complete solution.

3.3 FIVE PRINCIPLES

From the preceding rules, and indirectly from the criteria, five principles of software construction follow:

- The Linguistic Modular Units principle.
- The Self-Documentation principle.
- The Uniform Access principle.
- The Open-Closed principle.
- The Single Choice principle.

Linguistic Modular Units

The Linguistic Modular Units principle expresses that the formalism used to describe software at various levels (specifications, designs, implementations) must support the view of modularity retained:

Linguistic Modular Units principle

Modules must correspond to syntactic units in the language used.

The language mentioned may be a programming language, a design language, a specification language etc. In the case of programming languages, modules should be separately compilable.

What this principle excludes at any level — analysis, design, implementation — is combining a method that suggests a certain module concept and a language that does not offer the corresponding modular construct, forcing software developers to perform manual translation or restructuring. It is indeed not uncommon to see companies hoping to apply certain methodological concepts (such as modules in the Ada sense, or object-oriented principles) but then implement the result in a programming language such as Pascal or C which does not support them. Such an approach defeats several of the modularity criteria:

- Continuity: if module boundaries in the final text do not correspond to the logical decomposition of the specification or design, it will be difficult or impossible to maintain consistency between the various levels when the system evolves. A change of the specification may be considered small if it affects only a small number of specification modules; to ensure continuity, there must be a direct correspondence between specification, design and implementation modules.
- Direct Mapping: to maintain a clear correspondence between the structure of the model and the structure of the solution, you must have a clear syntactical identification of the conceptual units on both sides, reflecting the division suggested by your development method.
- Decomposability: to divide system development into separate tasks, you need to make sure that every task results in a well-delimited syntactic unit; at the implementation stage, these units must be separately compilable.
- Composability: how could we combine anything other than modules with unambiguous syntactic boundaries?
- Protection: you can only hope to control the scope of errors if modules are syntactically delimited.

Self-Documentation

Like the rule of Information Hiding, the Self-Documentation principle governs how we should document modules:

Self-Documentation principle

The designer of a module should strive to make all information about the module part of the module itself.

What this precludes is the common situation in which information about the module is kept in separate project documents.

The documentation under review here is **internal** documentation about components of the software, not **user** documentation about the resulting product, which may require separate products, whether paper, CD-ROM or Web pages — although, as noted in the discussion of software quality, one may see in the modern trend towards providing more and more on-line help a consequence of the same general idea.

"About documentation", page 14.

The most obvious justification for the Self-Documentation principle is the criterion of modular understandability. Perhaps more important, however, is the role of this

principle in helping to meet the continuity criterion. If the software and its documentation are treated as separate entities, it is difficult to guarantee that they will remain compatible — "in sync" — when things start changing. Keeping everything at the same place, although not a guarantee, is a good way to help maintain this compatibility.

Innocuous as this principle may seem at first, it goes against much of what the software engineering literature has usually suggested as good software development practices. The dominant view is that software developers, to deserve the title of software engineers, need to do what other engineers are supposed to: produce a kilogram of paper for every gram of actual deliverable. The encouragement to keep a record of the software construction process is good advice — but not the implication that software and its documentation are different products.

Such an approach ignores the specific property of software, which again and again comes back in this discussion: its changeability. If you treat the two products as separate, you risk finding yourself quickly in a situation where the documentation says one thing and the software does something else. If there is any worse situation than having no documentation, it must be having wrong documentation.

> A major advance of the past few years has been the appearance of *quality standards* for software, such as ISO certification, the "2167" standard and its successors from the US Department of Defense, and the Capability Maturity Model of the Software Engineering Institute. Perhaps because they often sprang out of models from other disciplines, they tend to specify a heavy paper trail. Several of these standards could have a stronger effect on software quality (beyond providing a mechanism for managers to cover their bases in case of later trouble) by enforcing the Self-Documentation principle.

"Using assertions for documentation: the short form of a class", page 390. See also chapter 23 and its last two exercises.

This book will draw on the Self-Documentation principle to define a method for documenting classes — the modules of object-oriented software construction — that includes the documentation of every module in the module itself. Not that the module *is* its documentation: there is usually too much detail in the software text to make it suitable as documentation (this was the argument for information hiding). Instead, the module should *contain* its documentation.

In this approach software becomes a single product that supports multiple **views**. One view, suitable for compilation and execution, is the full source code. Another is the abstract interface documentation of each module, enabling software developers to write client modules without having to learn the module's own internals, in accordance with the rule of Information Hiding. Other views are possible.

We will need to remember this rule when we examine the question of how to document the classes of object-oriented software construction.

Uniform Access

Also known as the Uniform Reference principle.

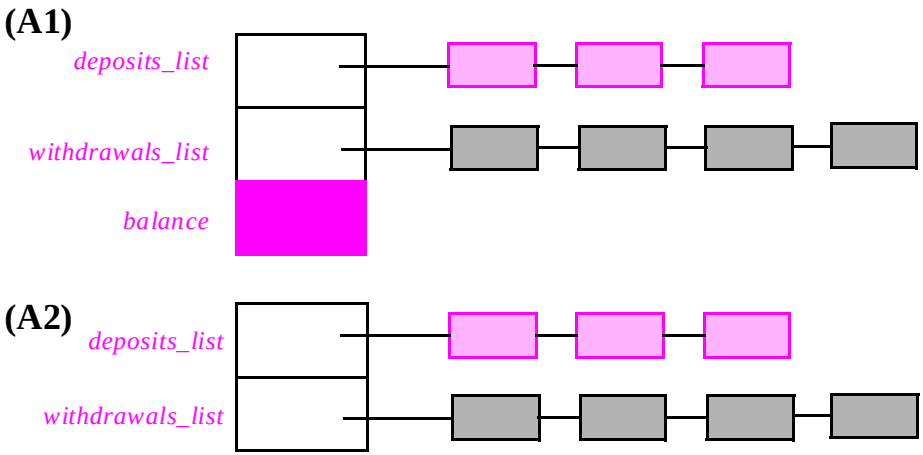
Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation. It follows from the Continuity criterion; you may also view it as a special case of Information Hiding.

Let *x* be a name used to access a certain data item (what will later be called an object) and *f* the name of a feature applicable to *x*. (A feature is an operation; this terminology will also be defined more precisely.) For example, *x* might be a variable representing a bank account, and *f* the feature that yields an account's current balance. Uniform Access addresses the question of how to express the result of applying *f* to *x*, using a notation that does not make any premature commitment as to how *f* is implemented.

In most design and programming languages, the expression denoting the application of *f* to *x* depends on what implementation the original software developer has chosen for feature *f*: is the value stored along with *x*, or must it be computed whenever requested? Both techniques are possible in the example of accounts and their balances:

A1 • You may represent the balance as one of the fields of the record describing each account, as shown in the figure. With this technique, every operation that changes the balance must take care of updating the *balance* field.

A2 • Or you may define a function which computes the balance using other fields of the record, for example fields representing the lists of withdrawals and deposits. With this technique the balance of an account is not stored (there is no *balance* field) but computed on demand.



Two representation for a bank account

A common notation, in languages such as Pascal, Ada, C, C++ and Java, uses *x*.*f* in case A1 and *f* (*x*) in case A2.

Choosing between representations A1 and A2 is a space-time tradeoff: one economizes on computation, the other on storage. The resolution of this tradeoff in favor of one of the solutions is typical of representation decisions that developers often reverse at least once during a project's lifetime. So for continuity's sake it is desirable to have a feature access notation that does not distinguish between the two cases; then if you are in charge of *x*'s implementation and change your mind at some stage, it will not be necessary to change the modules that use *f*. This is an example of the Uniform Access principle.

In its general form the principle may be expressed as:

Uniform Access principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Few languages satisfy this principle. An older one that did was Algol W, where both the function call and the access to a field were written $a\ (x)$. Object-oriented languages should satisfy Uniform Access, as did the first of them, Simula 67, whose notation is $x.f$ in both cases. The notation developed in part C will retain this convention.

The Open-Closed principle

Another requirement that any modular decomposition technique must satisfy is the Open-Closed principle:

Open-Closed principle

Modules should be both open and closed.

The contradiction between the two terms is only apparent as they correspond to goals of a different nature:

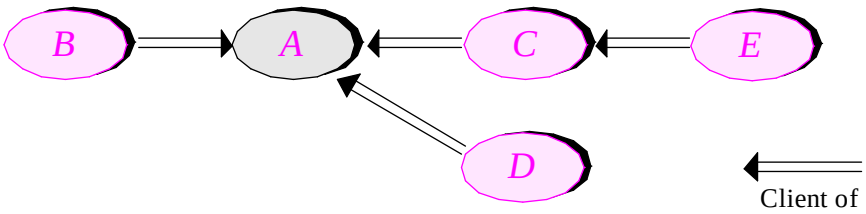
- A module is said to be open if it is still available for extension. For example, it should be possible to expand its set of operations or add fields to its data structures.
- A module is said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (its interface in the sense of information hiding). At the implementation level, closure for a module also implies that you may compile it, perhaps store it in a library, and make it available for others (its *clients*) to use. In the case of a design or specification module, closing a module simply means having it approved by management, adding it to the project's official repository of accepted software items (often called the project *baseline*), and publishing its interface for the benefit of other module authors.

The need for modules to be closed, and the need for them to remain open, arise for different reasons. Openness is a natural concern for software developers, as they know that it is almost impossible to foresee all the elements — data, operations — that a module will need in its lifetime; so they will wish to retain as much flexibility as possible for future changes and extensions. But it is just as necessary to close modules, especially from a project manager's viewpoint: in a system comprising many modules, most will depend on some others; a user interface module may depend on a parsing module (for parsing command texts) and on a graphics module, the parsing module itself may depend on a

lexical analysis module, and so on. If we never closed a module until we were sure it includes all the needed features, no multi-module software would ever reach completion: every developer would always be waiting for the completion of someone else's job.

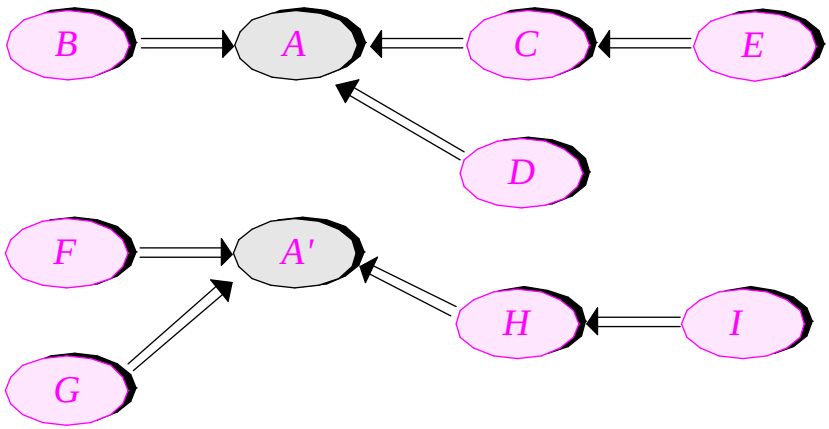
With traditional techniques, the two goals are incompatible. Either you keep a module open, and others cannot use it yet; or you close it, and any change or extension can trigger a painful chain reaction of changes in many other modules, which relied on the original module directly or indirectly.

The two figures below illustrate a typical situation where the needs for open and closed modules are hard to reconcile. In the first figure, module *A* is used by client modules *B*, *C*, *D*, which may themselves have their own clients (*E*, *F*, …).



A module and its clients

Later on, however, the situation is disrupted by the arrival of new clients — *B′* and others — which need an extended or adapted version of *A*, which we may call *A′*:



Old and new clients

With non-O-O methods, there seem to be only two solutions, equally unsatisfactory:

N1 • You may adapt module *A* so that it will offer the extended or modified functionality (*A′*) required by the new clients.

N2 • You may also decide to leave *A* as it is, make a copy, change the module's name to *A′* in the copy, and perform all the necessary adaptations on the new module. With this technique *A′* retains no further connection to *A*.

The potential for disaster with solution N1 is obvious. *A* may have been around for a long time and have many clients such as *B*, *C* and *D*. The adaptations needed to satisfy the new clients' requirements may invalidate the assumptions on the basis of which the old ones used *A*; if so the change to *A* may start a dramatic series of changes in clients, clients of clients and so on. For the project manager, this is a nightmare come true: suddenly, entire parts of the software that were supposed to have been finished and sealed off ages ago get reopened, triggering a new cycle of development, testing, debugging and documentation. If many a software project manager has the impression of living the Sisyphus syndrome — the impression of being sentenced forever to carry a rock to the top of the hill, only to see it roll back down each time — it is for a large part because of the problems caused by this need to reopen previously closed modules.

On the surface, solution N2 seems better: it avoids the Sisyphus syndrome since it does not require modifying any existing software (anything in the top half of the last figure). But in fact this solution may be even more catastrophic since it only postpones the day of reckoning. If you extrapolate its effects to many modules, many modification requests and a long period, the consequences are appalling: an explosion of variants of the original modules, many of them very similar to each other although never quite identical.

In many organizations, this abundance of modules, not matched by abundance of available functionality (many of the apparent variants being in fact quasi-clones), creates a huge *configuration management* problem, which people attempt to address through the use of complex tools. Useful as these tools may be, they offer a cure in an area where the first concern should be prevention. Better avoid redundancy than manage it.

Exercise E3.6, page 66, asks you to discuss how much need will remain for configuration management in an O-O context.

Configuration management will remain useful, of course, if only to find the modules which must be reopened after a change, and to avoid unneeded module recompilations.

But how can we have modules that are both open and closed? How can we keep *A* and everything in the top part of the figure unchanged, while providing *A'* to the bottom clients, and avoiding duplication of software? The object-oriented method will offer a particularly elegant contribution thanks to inheritance.

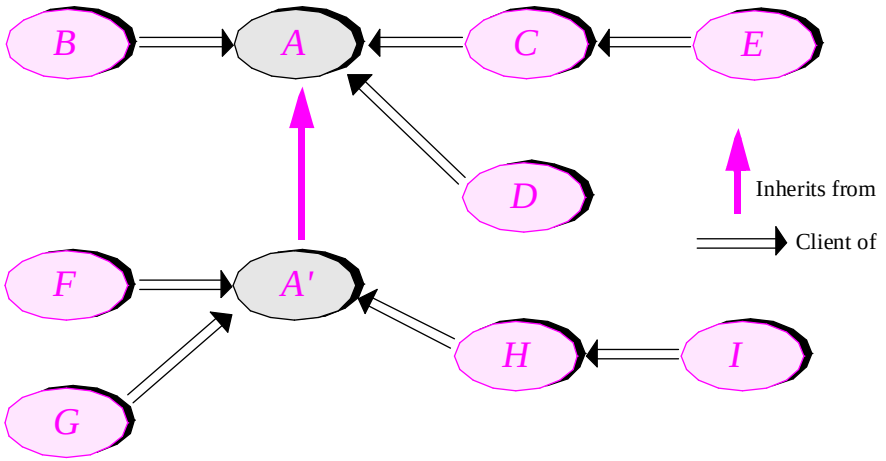
The detailed study of inheritance appears in later chapters, but here is a preview of the basic idea. To get us out of the *change or redo* dilemma, inheritance will allow us to define a new module *A'* in terms of an existing module *A* by stating the differences only. We will write *A'* as

```
class A' inherit
        A
                redefine f, g, … end
feature
        f is …
        g is …
        …
        u is …
        …
end
```

where the **feature** clause contains both the definition of the new features specific to *A'*, such as *u*, and the redefinition of those features (such as *f*, *g*, …) whose form in *A'* is different from the one they had in *A*.

The pictorial representation for inheritance will use an arrow from the heir (the new class, here *A'*) to the parent (here *A*):

Adapting a module to new clients



Thanks to inheritance, O-O developers can adopt a much more incremental approach to software development than used to be possible with earlier methods.

One way to describe the open-closed principle and the consequent object-oriented techniques is to think of them as a *organized hacking*. "Hacking" is understood here as a slipshod approach to building and modifying code (not in the more recent sense of breaking into computer networks, which, organized or not, no one should condone). The hacker may seem bad but often his heart is pure. He sees a useful piece of software, which is *almost* able to address the needs of the moment, more general than the software's original purpose. Spurred by a laudable desire not to redo what can be reused, our hacker starts modifying the original to add provisions for new cases. The impulse is good but the effect is often to pollute the software with many clauses of the form **if** *that_special_case* **then**…, so that after a few rounds of hacking, perhaps by a few different hackers, the software starts resembling a chunk of Swiss cheese that has been left outside for too long in August (if the tastelessness of this metaphor may be forgiven on the grounds that it does its best to convey the presence in such software of both holes and growth).

The organized form of hacking will enable us to cater to the variants without affecting the consistency of the original version.

A word of caution: nothing in this discussion suggests *dis*organized hacking. In particular:

- If you have control over the original software and can rewrite it so that it will address the needs of several kinds of client at no extra complication, you should do so.

- Neither the Open-Closed principle nor redefinition in inheritance is a way to address design flaws, let alone bugs. *If there is something wrong with a module, you should fix it* — not leave the original as it is and try to correct the problem in a derived module. (The only potential exception to this rule is the case of flawed software which you are not at liberty to modify.) The Open-Closed principle and associated techniques are intended for the adaptation of healthy modules: modules that, although they may not suffice for some new uses, meet their own well-defined requirements, to the satisfaction of their own clients.

Single Choice

The last of the five modularity principles may be viewed as a consequence of both the Open-Closed and Information Hiding rules.

Before examining the Single Choice principle in its full generality, let us look at a typical example. Assume you are building a system to manage a library (in the non-software sense of the term: a collection of books and other publications, not software modules). The system will manipulate data structures representing publications. You may have declared the corresponding type as follows in Pascal-Ada syntax:

```
type PUBLICATION =
        record
                author, title: STRING;
                publication_year: INTEGER
        case pubtype: (book, journal, conference_proceedings) of
                book: (publisher: STRING);
                journal: (volume, issue: STRING);
                proceedings: (editor, place: STRING)   -- Conference proceedings
        end
```

This particular form uses the Pascal-Ada notion of "record type with variants" to describe sets of data structures with some fields (here *author*, *title*, *publication_year*) common to all instances, and others specific to individual variants.

> The use of a particular syntax is not crucial here; Algol 68 and C provide an equivalent mechanism through the notion of union type. A union type is a type *T* defined as the union of pre-existing types *A*, *B*, …: a value of type *T* is either a value of type *A*, or a value of type *B*, … Record types with variants have the advantage of clearly associating a tag, here *book*, *journal*, *conference_proceedings*, with each variant.

Let *A* be the module that contains the above declaration or its equivalent using another mechanism. As long as *A* is considered open, you may add fields or introduce new variants. To enable *A* to have clients, however, you must close the module; this means that you implicitly consider that you have listed all the relevant fields and variants. Let *B* be a typical client of *A*. *B* will manipulate publications through a variable such as

```
p: PUBLICATION
```

and, to do just about anything useful with *p*, will need to discriminate explicitly between the various cases, as in:

case *p* **of**
book: … Instructions which may access the field *p.publisher* …
journal: … Instructions which may access fields *p.volume*, *p.issue* …
proceedings: … Instructions which may access fields *p.editor*, *p.place* …
end

The **case** instruction of Pascal and Ada comes in handy here; it is of course on purpose that its syntax mirrors the form of the declaration of a record type with variants. Fortran and C will emulate the effect through multi-target goto instructions (**switch** in C). In these and other languages a multi-branch conditional instruction (**if** … **then** … **elseif** … **elseif** … **else** … **end**) will also do the job.

Aside from syntactic variants, the principal observation is that to perform such a discrimination every client must know the exact list of variants of the notion of publication supported by *A*. The consequence is easy to foresee. Sooner or later, you will realize the need for a new variant, such as technical reports of companies and universities. Then you will have to extend the definition of type *PUBLICATION* in module *A* to support the new case. Fair enough: you have modified the conceptual notion of publication, so you should update the corresponding type declaration. This change is logical and inevitable. Far harder to justify, however, is the other consequence: any client of *A*, such as *B*, will also require updating if it used a structure such as the above, relying on an explicit list of cases for *p*. This may, as we have seen, be the case for most clients.

What we observe here is a disastrous situation for software change and evolution: a simple and natural addition may cause a chain reaction of changes across many client modules.

The issue will arise whenever a certain notion admits a number of variants. Here the notion was "publication" and its initial variants were book, journal article, conference proceedings; other typical examples include:

- In a graphics system: the notion of figure, with such variants as polygon, circle, ellipse, segment and other basic figure types.
- In a text editor: the notion of user command, with such variants as line insertion, line deletion, character deletion, global replacement of a word by another.
- In a compiler for a programming language, the notion of language construct, with such variants as instruction, expression, procedure.

In any such case, we must accept the possibility that the list of variants, although fixed and known at some point of the software's evolution, may later be changed by the addition or removal of variants. To support our long-term, software engineering view of the software construction process, we must find a way to **protect** the software's structure against the effects of such changes. Hence the Single Choice principle:

Single Choice principle

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

By requiring that knowledge of the list of choices be confined to just one module, we prepare the scene for later changes: if variants are added, we will only have to update the module which has the information — the point of single choice. All others, in particular its clients, will be able to continue their business as usual.

See "DYNAMIC BINDING", 14.4, page 480.

Once again, as the publications example shows, traditional methods do not provide a solution; once again, object technology will show the way, here thanks to two techniques connected with inheritance: polymorphism and dynamic binding. No sneak preview in this case, however; these techniques must be understood in the context of the full method.

The Single Choice principle prompts a few more comments:

- The number of modules that know the list of choices should be, according to the principle, exactly one. The modularity goals suggest that we want *at most one* module to have this knowledge; but then it is also clear that *at least one* module must possess it. You cannot write an editor unless at least one component of the system has the list of all supported commands, or a graphics system unless at least one component has the list of all supported figure types, or a Pascal compiler unless at least one component "knows" the list of Pascal constructs.

- Like many of the other rules and principles studied in this chapter, the principle is about **distribution of knowledge** in a software system. This question is indeed crucial to the search for extendible, reusable software. To obtain solid, durable system architectures you must take stringent steps to limit the amount of information available to each module. By analogy with the methods employed by certain human organizations, we may call this a **need-to-know** policy: barring every module from accessing any information that is not strictly required for its proper functioning.

See the second figure on page 58.

- You may view the Single Choice principle as a direct consequence of the Open-Closed principle. Consider the publications example in light of the figure that illustrated the need for open-closed modules: *A* is the module which includes the original declaration of type *PUBLICATION*; the clients *B*, *C*, … are the modules that relied on the initial list of variants; *A'* is the updated version of *A* offering an extra variant (technical reports).

- You may also understand the principle as a strong form of Information Hiding. The designer of supplier modules such as *A* and *A'* seeks to hide information (regarding the precise list of variants available for a certain notion) from the clients.

3.4 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The choice of a proper module structure is the key to achieving the aims of reusability and extendibility.
- Modules serve for both software decomposition (the top-down view) and software composition (bottom-up).
- Modular concepts apply to specification and design as well as implementation.
- A comprehensive definition of modularity must combine several perspectives; the various requirements may sometimes appear at odds with each other, as with decomposability (which encourages top-down methods) and composability (which favors a bottom-up approach).
- Controlling the amount and form of communication between modules is a fundamental step in producing a good modular architecture.
- The long-term integrity of modular system structures requires information hiding, which enforces a rigorous separation of interface and implementation.
- Uniform access frees clients from internal representation choices in their suppliers.
- A closed module is one that may be used, through its interface, by client modules.
- An open module is one that is still subject to extension.
- Effective project management requires support for modules that are both open and closed. But traditional approaches to design and programming do not permit this.
- The principle of Single Choice directs us to limit the dissemination of exhaustive knowledge about variants of a certain notion.

3.5 BIBLIOGRAPHICAL NOTES

The design method known as “structured design” [Yourdon 1979] emphasized the importance of modular structures. It was based on an analysis of module “cohesion” and “coupling”. But the view of modules implicit in structured design was influenced by the traditional notion of subroutine, which limits the scope of the discussion.

The principle of uniform access comes originally (under the name “uniform reference”) from [Geschke 1975].

The discussion of uniform access cited the Algol W language, a successor to Algol 60 and forerunner to Pascal (but offering some interesting mechanisms not retained in Pascal), designed by Wirth and Hoare and described in [Hoare 1966].

Information hiding was introduced in two milestone articles by David Parnas [Parnas 1972] [Parnas 1972a].

Configuration management tools that will recompile the modules affected by modifications in other modules, based on an explicit list of module dependencies, are based on the ideas of the Make tool, originally for Unix [Feldman 1979]. Recent tools — there are many on the market — have added considerable functionality to the basic ideas.

Some of the exercises below ask you to develop metrics to evaluate quantitatively the various informal measures of modularity developed in this chapter. For some results in O-O metrics, see the work of Christine Mingins [Mingins 1993] [Mingins 1995] and Brian Henderson-Sellers [Henderson-Sellers 1996a].

EXERCISES

E3.1 Modularity in programming languages

Examine the modular structures of any programming language which you know well and assess how they support the criteria and principles developed in this chapter.

E3.2 The Open-Closed principle (for Lisp programmers)

Many Lisp implementations associate functions with function names at run time rather than statically. Does this feature make Lisp more supportive of the Open-Closed principle than more static languages?

E3.3 Limits to information hiding

Can you think of circumstances where information hiding should *not* be applied to relations between modules?

E3.4 Metrics for modularity (term project)

The criteria, rules and principles of modularity of this chapter were all introduced through qualitative definitions. Some of them, however, may be amenable to quantitative analysis. The possible candidates include:

- Modular continuity.
- Few Interfaces.
- Small Interfaces.
- Explicit Interfaces.
- Information Hiding.
- Single Choice.

Explore the possibility of developing modularity metrics to evaluate how modular a software architecture is according to some of these viewpoints. The metrics should be size-independent: increasing the size of a system without changing its modular structure should not change its complexity measures. (See also the next exercise.)

E3.5 Modularity of existing systems

Apply the modularity criteria, rules and principles of this chapter to evaluate a system to which you have access. If you have answered the previous exercise, apply any proposed modularity metric.

Can you draw any correlations between the results of this analysis (qualitative, quantitative or both) and assessments of structural complexity for the systems under study, based either on informal analysis or, if available, on actual measurements of debugging and maintenance costs?

E3.6 Configuration management and inheritance

(This exercise assumes knowledge of inheritance techniques described in the rest of this book. It is not applicable if you have read this chapter as part of a first, sequential reading of the book.)

The discussion of the open-closed principle indicated that in non-object-oriented approaches the absence of inheritance places undue burden on configuration management tools, since the desire to avoid reopening closed modules may lead to the creation of too many module variants. Discuss what role remains for configuration management in an object-oriented environment where inheritance *is* present, and more generally how the use of object technology affects the problem of configuration management.

If you are familiar with specific configuration management tools, discuss how they interact with inheritance and other principles of O-O development.