5

Towards object technology

Extendibility, reusability and reliability, our principal goals, require a set of conditions defined in the preceding chapters. To achieve these conditions, we need a systematic method for decomposing systems into modules.

This chapter presents the basic elements of such a method, based on a simple but far-reaching idea: build every module on the basis of some object type. It explains the idea, develops the rationale for it, and explores some of the immediate consequences.

A word of warning. Given today's apparent prominence of object technology, some readers might think that the battle has been won and that no further rationale is necessary. This would be a mistake: we need to understand the basis for the method, if only to avoid common misuses and pitfalls. It is in fact frequent to see the word "object-oriented" (like "structured" in an earlier era) used as mere veneer over the most conventional techniques. Only by carefully building the case for object technology can we learn to detect improper uses of the buzzword, and stay away from common mistakes reviewed later in this chapter.

5.1 THE INGREDIENTS OF COMPUTATION

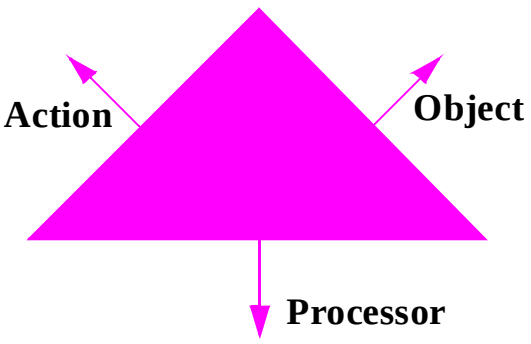
The crucial question in our search for proper software architectures is *modularization*: what criteria should we use to find the modules of our software?

To obtain the proper answer we must first examine the contending candidates.

The basic triangle

Three forces are at play when we use software to perform some computations:

The three forces of computation



To execute a software system is to use certain *processors* to apply certain *actions* to certain *objects*.

The processors are the computation devices, physical or virtual, that execute instructions. A processor can be an actual processing unit (the CPU of a computer), a process on a conventional operating system, or a "thread" if the OS is multi-threaded.

The actions are the operations making up the computation. The exact form of the actions that we consider will depend on the level of granularity of our analysis: at the hardware level, actions are machine language operations; at the level of the hardware-software machine, they are instructions of the programming language; at the level of a software system, we can treat each major step of a complex algorithm as a single action.

The objects are the data structures to which the actions apply. Some of these objects, the data structures built by a computation for its own purposes, are internal and exist only while the computation proceeds; others (contained in the files, databases and other persistent repositories) are external and may outlive individual computations.

Processors will become important when we discuss **concurrent** forms of computation, in which several sub-computations can proceed in parallel; then we will need to consider two or more processors, physical or virtual. But that is the topic of a later chapter; for the moment we can limit our attention to non-concurrent, or *sequential* computations, relying on a single processor which will remain implicit.

Concurrency is the topic of chapter 30.

This leaves us with actions and objects. The duality between actions and objects — what a system does *vs.* what it does it to — is a pervasive theme in software engineering.

A note of terminology. Synonyms are available to denote each of the two aspects: the word *data* will be used here as a synonym for *objects*; for *action* the discussion will often follow common practice and talk about the *functions* of a system.

The term "function" is not without disadvantages, since software discussions also use it in at least two other meanings: the mathematical sense, and the programming sense of subprogram returning a result. But we can use it without ambiguity in the phrase *the functions of a system*, which is what we need here.

The reason for using this word rather than "action" is the mere grammatical convenience of having an associated adjective, used in the phrase *functional decomposition*. "Action" has no comparable derivation. Another term whose meaning is equivalent to that of "action" for the purpose of this discussion is *operation*.

Any discussion of software issues must account for both the object and function aspects; so must the design of any software system. But there is one question for which we must choose — the question of this chapter: what is the appropriate criterion for finding the modules of a system? Here we must decide whether modules will be built as units of functional decomposition, or around major types of objects.

From the answer will follow the difference between the object-oriented approach and other methods. Traditional approaches build each module around some unit of functional decomposition — a certain piece of the action. The object-oriented method, instead, builds each module around some type of objects.

This book, predictably, develops the latter approach. But we should not just embrace O-O decomposition because the title of the book so implies, or because it is the "in" thing to do. The next few sections will carefully examine the arguments that justify using object types as the basis for modularization — starting with an exploration of the merits and limitations of traditional, non-O-O methods. Then we will try to get a clearer understanding of what the word "object" really means for software development, although the full answer, requiring a little theoretical detour, will only emerge in the next chapter.

We will also have to wait until the next chapter for the final settlement of the formidable and ancient fight that provides the theme for the rest of the present discussion: the War of the Objects and the Functions. As we prepare ourselves for a campaign of slander against the functions as a basis for system decomposition, and of corresponding praise for the objects, we must not forget the observation made above: in the end, our solution to the software structuring problem must provide space for both functions and objects — although not necessarily on an equal basis. To discover this new world order, we will need to define the respective roles of its first-class and second-class citizens.

5.2 FUNCTIONAL DECOMPOSITION

We should first examine the merits and limitations of the traditional approach: using functions as a basis for the architecture of software systems. This will not only lead us to appreciate why we need something else — object technology — but also help us avoid, when we do move into the object world, certain methodological pitfalls such as premature operation ordering, which have been known to fool even experienced O-O developers.

Continuity

"Modular continuity", page 44.

A key element in answering the question "should we structure systems around functions or around data?" is the problem of extendibility, and more precisely the goal called *continuity* in our earlier discussions. As you will recall, a design method satisfies this criterion if it yields stable architectures, keeping the amount of design change commensurate with the size of the specification change.

Continuity is a crucial concern if we consider the real lifecycle of software systems, including not just the production of an acceptable initial version, but a system's long-term evolution. Most systems undergo numerous changes after their first delivery. Any model of software development that only considers the period leading to that delivery and ignores the subsequent era of change and revision is as remote from real life as those novels which end when the hero marries the heroine — the time which, as everyone knows, marks the beginning of the really interesting part.

To evaluate the quality of an architecture (and of the method that produced it), we should not just consider how easy it was to obtain this architecture initially: it is just as important to ascertain how well the architecture will weather change.

Top-down design was sketched in "Modular decomposability", page 40.

The traditional answer to the question of modularization has been top-down functional decomposition, briefly introduced in an earlier chapter. How well does top-down design respond to the requirements of modularity?

Top-down development

> *There was a most ingenious architect who had contrived a new method for building houses, by beginning at the roof, and working downwards to the foundation, which he justified to me by the like practice of those two prudent insects, the bee and the spider.*
>
> Jonathan Swift: *Gulliver's Travels*, Part III, *A Voyage to Laputa, etc.*, Chapter 5.

The top-down approach builds a system by stepwise refinement, starting with a definition of its abstract function. You start the process by expressing a topmost statement of this function, such as

[C0]

"Translate a C program to machine code"

or:

[P0]

"Process a user command"

and continue with a sequence of refinement steps. Each step must decrease the level of abstraction of the elements obtained; it decomposes every operation into a combination of one or more simpler operations. For example, the next step in the first example (the C compiler) could produce the decomposition

[C1]

"Read program and produce sequence of tokens"
"Parse sequence of tokens into abstract syntax tree"
"Decorate tree with semantic information"
"Generate code from decorated tree"

or, using an alternative structure (and making the simplifying assumption that a C program is a sequence of function definitions):

[C'1]

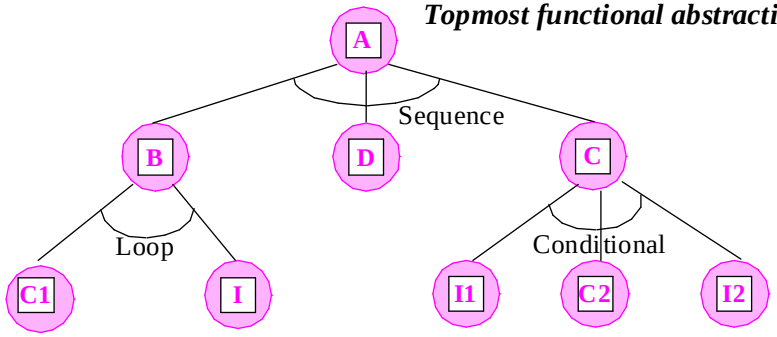
from
    "Initialize data structures"
until
    "All function definitions processed"
loop
    "Read in next function definition"
    "Generate partial code"
end
"Fill in cross references"

In either case, the developer must at each step examine the remaining incompletely expanded elements (such as "Read program …" and "All function definitions processed") and expand them, using the same refinement process, until everything is at a level of abstraction low enough to allow direct implementation.

We may picture the process of top-down refinement as the development of a tree. Nodes represent elements of the decomposition; branches show the relation "*B* is part of the refinement of *A*".

Top-down design: tree structure

(*This figure first appeared on page 41.*)



The top-down approach has a number of advantages. It is a logical, well-organized thought discipline; it can be taught effectively; it encourages orderly development of systems; it helps the designer find a way through the apparent complexity that systems often present at the initial stages of their design.

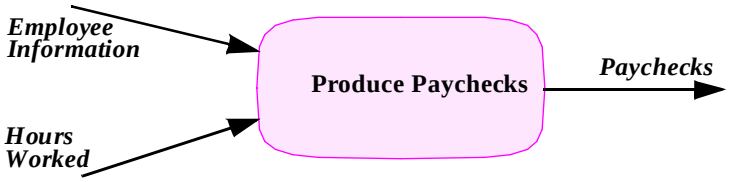
The top-down approach can indeed be useful for developing individual algorithms. But it also suffers from limitations that make it questionable as a tool for the design of entire systems:

- The very idea of characterizing a system by just one function is subject to doubt.
- By using as a basis for modular decomposition the properties that tend to change the most, the method fails to account for the evolutionary nature of software systems.

Not just one function

In the evolution of a system, what may originally have been perceived as the system's main function may become less important over time.

Consider a typical payroll system. When stating his initial requirement, the customer may have envisioned just what the name suggests: a system to produce paychecks from the appropriate data. His view of the system, implicit or explicit, may have been a more ambitious version of this:



Structure of a simple payroll program

The system takes some inputs (such as record of hours worked and employee information) and produces some outputs (paychecks and so on). This is a simple enough functional specification, in the strict sense of the word functional: it defines the program as a mechanism to perform one function — pay the employees. The top-down functional method is meant precisely for such well-defined problems, where the task is to perform a single function — the "top" of the system to be built.

Assume, however, that the development of our payroll program is a success: the program does the requisite job. Most likely, the development will not stop there. Good systems have the detestable habit of giving their users plenty of ideas about all the other things they could do. As the system's developer, you may initially have been told that all you had to do was to generate paychecks and a few auxiliary outputs. But now the requests for extensions start landing on your desk: Could the program gather some statistics on the side? I did tell you that next quarter we are going to start paying some employees monthly and others biweekly, did I not? And, by the way, I need a summary every month for management, and one every quarter for the shareholders. The accountants want their own output for tax preparation purposes. Also, you are keeping all this salary information, right? It would be really nifty to let Personnel access it interactively. I cannot imagine why that would be a difficult functionality to add.

This phenomenon of having to add unanticipated functions to successful systems occurs in all application areas. A nuclear code that initially just applied some algorithm to produce tables of numbers from batch input will be extended to handle graphical input and output or to maintain a database of previous results. A compiler that just translated valid source into object code will after a while double up as a syntax verifier, a static analyzer, a pretty-printer, even a programming environment.

This change process is often incremental. The new requirements evolve from the initial ones in a continuous way. The new system is still, in many respects, "the same system" as the old one: still a payroll system, a nuclear code, a compiler. But the original "main function", which may have seemed so important at first, often becomes just one of many functions; sometimes, it just vanishes, having outlived its usefulness.

If analysis and design have used a decomposition method based on the function, the system structure will follow from the designers' original understanding of the system's main function. As the system evolves, the designers may feel sorry (or its maintainers, if different people, may feel angry) about that original assessment. Each addition of a new function, however incremental it seems to the customer, risks invalidating the entire structure.

It is crucial to find, as a criterion for decomposition, properties less volatile than the system's main function.

Finding the top

Top-down methods assume that every system is characterized, at the most abstract level, by its main function. Although it is indeed easy to specify textbook examples of algorithmic problems — the Tower of Hanoi, the Eight Queens and the like — through their functional "tops", a more useful description of practical software systems considers each of them as offering a number of services. Defining such a system by a single function is usually possible, but yields a rather artificial view.

Take an operating system. It is best understood as a system that provides certain services: allocating CPU time, managing memory, handling input and output devices, decoding and carrying out users' commands. The modules of a well-structured OS will tend to organize themselves around these groups of functions. But this is not the architecture that you will get from top-down functional decomposition; the method forces you, as the designer, to answer the artificial question "what is the topmost function?", and then to use the successive refinements of the answer as a basis for the structure. If hard pressed you could probably come up with an initial answer of the form

"Process all user requests"

which you could then refine into something like

```
from
        boot
until
        halted or crashed
loop
        "Read in a user's request and put it into input queue"
        "Get a request r from input queue"
        "Process r"
        "Put result into output queue"
        "Get a result o from output queue"
        "Output o to its recipient"
end
```

Refinements can go on. From such premises, however, it is unlikely that anyone can ever develop a reasonably structured operating system.

Even systems which may at first seem to belong to the "one input, one abstract function, one output" category reveal, on closer examination, a more diverse picture. Consider the earlier example of a compiler. Reduced to its bare essentials, or to the view of older textbooks, a compiler is the implementation of one input-to-output function: transforming source text in some programming language into machine code for a certain platform. But that is not a sufficient view of a modern compiler. Among its many services, a compiler will perform error detection, program formating, some configuration management, logging, report generation.

Another example is a typesetting program, taking input in some text processing format — TEX, Microsoft Word, FrameMaker … — and generating output in HTML, Postscript or Adobe Acrobat format. Again we may view it at first as just an input-to-output filter. But most likely it will perform a number of other services as well, so it seems more interesting, when we are trying to characterize the system in the most general way, to consider the various types of data it manipulates: documents, chapters, sections, paragraphs, lines, words, characters, fonts, running heads, titles, figures and others.

The seemingly obvious starting point of top-down design — the view that each new development fulfills a request for a specific function — is subject to doubt:

Real systems have no top.

Functions and evolution

Not only is the main function often not the best criterion to characterize a system initially: it may also, as the system evolves, be among the first properties to change, forcing the top-down designer into frequent redesign and defeating our attempts to satisfy the continuity requirement.

Consider the example of a program that has two versions, a "batch" one which handles every session as a single big run over the problem, and an interactive one in which a session is a sequence of transactions, with a much finer grain of user-system communication. This is typical of large scientific programs, which often have a "let it run a big chunk of computation for the whole night" version and a "let me try out a few things and see the results at once then continue with something else" version.

The top-down refinement of the batch version might begin as

[B0] -- Top-level abstraction

"Solve a complete instance of the problem"

[B1] -- First refinement

"Read input values"

"Compute results"

"Output results"

and so on. The top-down development of the interactive version, for its part, could proceed in the following style:

```
[I1]
        "Process one transaction"
[I2]
        if "New information provided by the user" then
                "Input information"
                "Store it"
        elseif "Request for information previously given" then
                "Retrieve requested information"
                "Output it"
        elseif "Request for result" then
                if "Necessary information available" then
                        "Retrieve requested result"
                        "Output it"
                else
                        "Ask for confirmation of the request"
                        if Yes then
                                "Obtain required information"
                                "Compute requested result"
                                "Output result"
                        end
                end
        else
                (Etc.)
```

Started this way, the development will yield an entirely different result. The top-down approach fails to account for the property that the final programs are but two different versions of the same software system — whether they are developed concurrently or one has evolved from the other.

This example brings to light two of the most unpleasant consequences of the top-down approach: its focus on the external interface (implying here an early choice between batch and interactive) and its premature binding of temporal relations (the order in which actions will be executed).

Interfaces and software design

System architecture should be based on substance, not form. But top-down development tends to use the most superficial aspect of the system — its external interface — as a basis for its structure.

The focus on external interfaces is inevitable in a method that asks "What will the system do for the end user?" as the key question: the answer will tend to emphasize the most external aspects.

The user interface is only one of the components of a system. Often, it is also among the most volatile, if only because of the difficulty of getting it right the first time; initial versions may be of the mark, requiring experimentation and user feedback to obtain a satisfactory solution. A healthy design method will try to separate the interface from the rest of the system, using more stable properties as the basis for system structuring.

It is in fact often possible to build the interface separately from the rest of the system, using one of the many tools available nowadays to produce elegant and user-friendly interfaces, often based on object-oriented techniques. The user interface then becomes almost irrelevant to the overall system design.

Chapter 32 discusses techniques and tools for user interfaces.

Premature ordering

The preceding examples illustrate another drawback of top-down functional decomposition: premature emphasis on temporal constraints. Each refinement expands a piece of the abstract structure into a more detailed *control* architecture, specifying the order in which various functions (various pieces of the action) will be executed. Such ordering constraints become essential properties of the system architecture; but they too are subject to change.

Recall the two alternative candidate structures for the first refinement of a compiler:

[C1]

```
"Read program and produce sequence of tokens"
"Parse sequence of tokens into abstract syntax tree"
"Decorate tree with semantic information"
"Generate code from decorated tree"
```

[C'1]

```
from
        "Initialize data structures"
until
        "All function definitions processed"
loop
        "Read in next function definition"
        "Generate partial code"
end

"Fill in cross references"
```

As in the preceding example we start with two completely different architectures. Each is defined by a control structure (a sequence of instructions in the first case, a loop followed by an instruction in the second), implying strict ordering constraints between the elements of the structure. But freezing such ordering relations at the earliest stages of design is not reasonable. Issues such as the number of passes in a compiler and the sequencing of various activities (lexical analysis, parsing, semantic processing, optimization) have many possible solutions, which the designers must devise by considering space-time tradeoffs and other criteria which they do not necessarily master

at the beginning of a project. They can perform fruitful design and implementation work on the components long before freezing their temporal ordering, and will want to retain this sequencing freedom for as long as possible. Top-down functional design does not provide such flexibility: you must specify the order of executing operations before you have had a chance to understand properly what these operations will do.

See the bibliographical notes for references on the methods cited.

Some design methods that attempt to correct some of the deficiencies of functional top-down design also suffer from this premature binding of temporal relationships. This is the case, among others, with the dataflow-directed method known as structured analysis and with Merise (a method popular in some European countries).

Object-oriented development, for its part, stays away from premature ordering. The designer studies the various operations applicable to a certain kind of data, and specifies the effect of each, but defers for as long as possible specifying the operations' order of execution. This may be called the **shopping list** approach: list needed operations — all the operations that you may need; ignore their ordering constraints until as late as possible in the software construction process. The result is much more extendible architectures.

Ordering and O-O development

The observations on the risks of premature ordering deserve a little more amplification because even object-oriented designers are not immune. The shopping list approach is one of the least understood parts of the method and it is not infrequent to see O-O projects fall into the old trap, with damaging effects on quality. This can result in particular from misuse of the *use case* idea, which we will encounter in the study of O-O methodology.

Chapter 11 presents assertions.

The problem is that the order of operations may seem so obvious a property of a system that it will weasel itself into the earliest stages of its design, with dire consequences if it later turns out to be not so final after all. The alternative technique (under the "shopping list" approach), perhaps less natural at first but much more flexible, uses logical rather than temporal constraints. It relies on the assertion concept developed later in this book; we can get the basic idea now through a simple non-software example.

Consider the problem of buying a house, reduced (as a gross first approximation) to three operations: finding a house that suits you; getting a loan; signing the contract. With a method focusing on ordering we will describe the design as a simple sequence of steps:

[H]

find_house
get_loan
sign_contract

In the shopping list approach of O-O development we will initially refuse to attach too much importance to this ordering property. But of course constraints exist between the operations: you cannot sign a contract unless (let us just avoid saying *until* for the time being!) you have a desired house and a loan. We can express these constraints in logical rather than temporal form:

[H'1]

```
find_property
        ensure
                property_found

get_loan
        ensure
                loan_approved

sign_contract
        require
                property_found and loan_approved
```

The notation will only be introduced formally in chapter 11, but it should be clear enough here: **require** states a precondition, a logical property that an operation requires for its execution; and **ensure** states a postcondition, a logical property that will follow from an operation's execution. We have expressed that each of the first two operations achieves a certain property, and that the last operation requires both of these properties.

Why is the logical form of stating the constraints, H'1, better than the temporal form, H1? The answer is clear: H'1 expresses the minimum requirements, avoiding the overspecification of H1. And indeed H1 is too strong, as it rules out the scheme in which you get the loan first and then worry about the property — not at all absurd for a particular buyer whose main problem is financing. Another buyer might prefer the reverse order; we should support both schemes as long as they observe the logical constraint.

Now imagine that we turn this example into a realistic model of the process with the many tasks involved — title search, termite inspection, pre-qualifying for the loan, finding a real estate agent, selling your previous house if applicable, inviting your friends to the house-warming party… It may be possible to express the ordering constraints, but the result will be complicated and probably fragile (you may have to reconsider everything if you later include another task). The logical constraint approach scales up much more smoothly; each operation simply states what it needs and what it guarantees, all in terms of abstract properties.

Exercise E6.7, page 162 (in the next chapter).

These observations are particularly important for the would-be object designer, who may still be influenced by functional ideas, and might be tempted to rely on early identification of system usage scenarios ("use cases") as a basis for analysis. This is incompatible with object-oriented principles, and often leads to top-down functional decomposition of the purest form — even when the team members are convinced that they are using an object-oriented method.

We will examine, in our study of O-O methodological principles, what role can be found for use cases in object-oriented software construction.

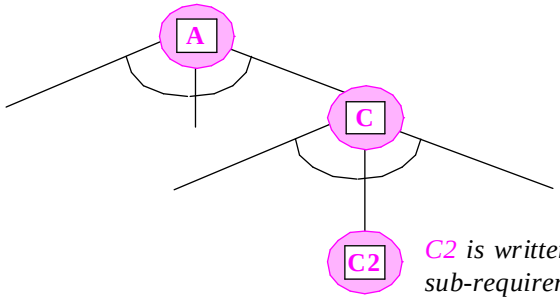
"Use cases", page 738.

Reusability

After this short advance incursion into the fringes of object territory, let us resume our analysis of the top-down method, considering it this time in relation to one of our principal goals, reusability.

Working top-down means that you develop software elements in response to particular subspecifications encountered in the tree-like development of a system. At a given point of the development, corresponding to the refinement of a certain node, you will detect the need for a specific function — such as analyzing an input command line — and write down its specification, which you or someone else will then implement.

The context of a module in top-down design



C2 is written to satisfy a sub-requirement of C.

The figure, which shows part of a top-down refinement tree, illustrates this property: *C2* is written to satisfy some sub-requirement of *C*; but the characteristics of *C2* are entirely determined by its immediate context — the needs of *C*. For example, *C* could be a module in charge of analyzing some user input, and *C2* could be the module in charge of analyzing one line (part of a longer input).

This approach is good at ensuring that the design will meet the initial specification, but it does not promote reusability. Modules are developed in response to specific subproblems, and tend to be no more general than implied by their immediate context. Here if *C* is meant for input texts of a specific kind, it is unlikely that *C2*, which analyzes one line of those texts, will be applicable to any other kind of input.

One can in principle include the concern for extendibility and generality in a top-down design process, and encourage developers to write modules that transcend the immediate needs which led to their development. But nothing in the method encourages generalization, and in practice it tends to produce modules with narrow specifications.

The very notion of top-down design suggests the reverse of reusability. Designing for reusability means building components that are as general as possible, then combining them into systems. This is a bottom-up process, at the opposite of the top-down idea of starting with the definition of "the problem" and deriving a solution through successive refinements.

On the project and product culture see [M 1995].

This discussion makes top-down design appear as a byproduct of what we can call the *project culture* in software engineering: the view that the unit of discourse is the individual project, independently of earlier and later projects. The reality is less simple: project *n* in a company is usually a variation on project *n – 1*, and a preview of project *n + 1*. By focusing on just one project, top-down design ignores this property of practical software construction,

Production and description

One of the reasons for the original attraction of top-down ideas is that a top-down style may be convenient to explain a design once it is in place. But what is good to document an existing design is not necessarily the best way to produce designs. This point was eloquently argued by Michael Jackson in *System Development*:

Quotation from [Jackson 1983], pages 370-371.

> *Top-down is a reasonable way of describing things which are already fully understood... But top-down is not a reasonable way of developing, designing, or discovering anything. There is a close parallel with mathematics. A mathematical textbook describes a branch of mathematics in a logical order: each theorem stated and proved is used in the proofs of subsequent theorems. But the theorems were not developed or discovered in this way, or in this order...*
>
> *When the developer of a system, or of a program, already has a clear idea of the completed result in his mind, he can use top-down to describe on paper what is in his head. This is why people can believe that they are performing top-down design or development, and doing so successfully: they confuse the method of description with the method of development... When the top-down phase begins, the problem is already solved, and only details remain to be solved.*

Top-down design: an assessment

This discussion of top-down functional design shows the method to be poorly adapted to the development of significant systems. It remains a useful paradigm for small programs and individual algorithms; it is certainly a helpful technique to *describe* well-understood algorithms, especially in programming courses. But it does not scale up to large practical software. By developing a system top-down you trade short-term convenience for long-term inflexibility; you unduly privilege one function over the others; you may be led to devoting your attention to interface characteristics at the expense of more fundamental properties; you lose sight of the data aspect; and you risk sacrificing reusability.

5.3 OBJECT-BASED DECOMPOSITION

The case for using objects (or more precisely, as seen below, object types) as the key to system modularization is based on the quality aims defined in chapter 1, in particular extendibility, reusability and compatibility.

The plea for using objects will be fairly short, since the case has already been made at least in part: many of the arguments against top-down, function-based design reappear naturally as evidence in favor of bottom-up, object-based design.

This evidence should not, however, lead us to dismiss the functions entirely. As noted at the beginning of this chapter, no approach to software construction can be complete unless it accounts for both the function and object parts. So we will need to retain a clear role for functions in the object-oriented method, even if they must submit to the

objects in the resulting system architectures. The notion of abstract data type will provide us with a definition of objects which reserves a proper place for the functions.

Extendibility

If the functions of a system, as discussed above, tend to change often over the system's life, can we find a more stable characterization of its essential properties, so as to guide our choice of modules and meet the goal of continuity?

The types of objects manipulated by the system are more promising candidates. Whatever happens to the payroll processing system used earlier as an example, it likely will still manipulate objects representing employees, salary scales, company regulations, hours worked, pay checks. Whatever happens to a compiler or other language processing tool, it likely will still manipulate source texts, token sequences, parse trees, abstract syntax trees, target code. Whatever happens to a finite element system, it likely will still manipulate matrices, finite elements and grids.

This argument is based on pragmatic observation, not on a proof that object types are more stable than functions. But experience seems to support it overwhelmingly.

The argument only holds if we take a high-level enough view of objects. If we understood objects in terms of their physical representations, we would not be much better off than with functions — as a matter of fact probably worse, since a top-down functional decomposition at least encourages abstraction. So the question of finding a suitably abstract description of objects is crucial; it will occupy all of the next chapter.

Reusability

The discussion of reusability pointed out that a routine (a unit of functional decomposition) was usually not sufficient as a unit of reusability.

See "Factoring Out Common Behaviors", page 85.

The presentation used a typical example: table searching. Starting with a seemingly natural candidate for reuse, a searching routine, it noted that we cannot easily reuse such a routine separately from the other operations that apply to a table, such as creation, insertion and deletion; hence the idea that a satisfactory reusable module for such a problem should be a collection of such operations. But if we try to understand the conceptual thread that unites all these operations, we find the type of objects to which they apply — tables.

Such examples suggest that object types, fully equipped with the associated operations, will provide stable units of reuse.

Compatibility

Another software quality factor, compatibility, was defined as the ease with which software products (for this discussion, modules) can be combined with each other.

It is difficult to combine actions if the data structures they access are not designed for that purpose. Why not instead try to combine entire data structures?

5.4 OBJECT-ORIENTED SOFTWARE CONSTRUCTION

We have by now accumulated enough background to consider a tentative definition of object-oriented software construction. This will only be a first attempt; a more concrete definition will follow from the discussion of abstract data types in the next chapter.

See page 147 for the final definition.

Object-oriented software construction (definition 1)

Object-oriented software construction is the software development method which bases the architecture of any software system on modules deduced from the types of objects it manipulates (rather than the function or functions that the system is intended to ensure).

An informal characterization of this approach may serve as a motto for the object-oriented designer:

OBJECT MOTTO

***Ask not first what the system does*:**

***Ask what it does it to*!**

To get a working implementation, you will of course, sooner or later, have to find out what it does. Hence the word *first*. Better later than sooner, says object-oriented wisdom. In this approach, the choice of main function is one of the very last steps to be taken in the process of system construction.

The developers will stay away, as long as possible, from the need to describe and implement the topmost function of the system. Instead, they will analyze the types of objects of the system. System design will progress through the successive improvements of their understanding of these object classes. It is a bottom-up process of building robust and extendible solutions to parts of the problem, and combining them into more and more powerful assemblies — until the final assembly which yields a solution of the original problem but, everyone hopes, is not the *only* possible one: the same components, assembled differently and probably combined with others, should be general enough to yield as a byproduct, if you have applied the method well and enjoyed your share of good luck, solutions to future problems as well.

For many software people this change in viewpoint is as much of a shock as may have been for others, in an earlier time, the idea of the earth orbiting around the sun rather than the reverse. It is also contrary to much of the established software engineering wisdom, which tends to present system construction as the fulfillment of a system's function as expressed in a narrow, binding requirements document. Yet this simple idea — look at the data first, forget the immediate purpose of the system — may hold the key to reusability and extendibility.

5.5 ISSUES

The above definition provides a starting point to discuss the object-oriented method. But besides providing components of the answer it also raises many new questions, such as:

- How to find the relevant object types.
- How to describe the object types.
- How to describe the relations and commonalities between object types.
- How to use object types to structure software.

The rest of this book will address these issues. Let us preview a few answers.

Finding the object types

See chapter 22.

The question "how shall we find the objects?" can seem formidable at first. A later chapter will examine it in some detail (in its more accurate version, which deals with object *types* rather than individual objects) but it is useful here to dispel some of the possible fears. The question does not necessarily occupy much of the time of experienced O-O developers, thanks in part to the availability of three sources of answers:

- Many objects are there just for the picking. They directly model objects of the physical reality to which the software applies. One of the particular strengths of object technology is indeed its power as a modeling tool, using software object types (classes) to model physical object types, and the method's inter-object-type relations (client, inheritance) to model the relations that exist between physical object types, such as aggregation and specialization. It does not take a treatise on object-oriented analysis to convince a software developer that a call monitoring system, in a telecommunications application, will have a class *CALL* and a class *LINE*, or that a document processing system will have a class *DOCUMENT*, a class *PARAGRAPH* and a class *FONT*.
- A source of object types is reuse: classes previously developed by others. This technique, although not always prominent in the O-O analysis literature, is often among the most useful in practice. We should resist the impulse to invent something if the problem has already been solved satisfactorily by others.
- Finally, experience and imitation also play a role. As you become familiar with successful object-oriented designs and design patterns (such as some of those described in this book and the rest of the O-O literature), even those which are not directly reusable in your particular application, you will be able to gain inspiration from these earlier efforts.

We will be in a much better position to understand these object-finding techniques and others once we have gained a better technical insight into the software notion of object — not to be confused with the everyday meaning of the word.

Describing types and objects

A question of more immediate concern, assuming we know how to obtain the proper object types to serve as a basis for modularizing our systems, is how to describe these types and their objects.

Two criteria must guide us in answering this question:

- The need to provide representation-independent descriptions, for fear of losing (as noted) the principal benefit of top-down functional design: abstraction.
- The need to re-insert the functions, giving them their proper place in software architectures whose decomposition is primarily based on the analysis of object types since (as also noted) we must in the end accommodate both aspects of the object-function duality.

The next chapter develops an object description technique achieving these goals.

Describing the relations and structuring software

Another question is what kind of relation we should permit between object types; since the modules will be based on object types, the answer also determines the structuring techniques that will be available to make up software systems from components.

In the purest form of object technology, only two relations exist: client and inheritance. They correspond to different kinds of possible dependency between two object types *A* and *B*:

- *B* is a client of *A* if every object of type *B* may contain information about one or more objects of type *A*.
- *B* is an heir of *A* if *B* denotes a specialized version of *A*.

Some widely used approaches to analysis, in particular information modeling approaches such as entity-relationship modeling, have introduced rich sets of relations to describe the many possible connections that may exist between the element of a system. To people used to such approaches, having to do with just two kinds of relation often seems restrictive at first. But this impression is not necessarily justified:

- The client relation is broad enough to cover many different forms of dependency. Examples include what is often called aggregation (the presence in every object of type *B* of a subobject of type *A*), reference dependency, and generic dependency.
- The inheritance relation covers specialization in its many different forms.
- Many properties of dependencies will be expressed in a more general form through other techniques. For example, to describe a 1-to-*n* dependency (every object of type *B* is connected to at least one and at most *n* objects of type *A*) we will express that *B* is a client of *A*, and include a **class invariant** specifying the exact nature of the client relation. The class invariant, being expressed in the language of logic, covers many more cases than the finite set of primitive relations offered by entity-relationship modeling or similar approaches.

5.6 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Computation involves three kinds of ingredient: processors (or threads of control), actions (or functions), and data (or objects).
- A system's architecture may be obtained from the functions or from the object types.
- A description based on object types tends to provide better stability over time and better reusability than one based on an analysis of the system's functions.
- It is usually artificial to view a system as consisting of just one function. A realistic system usually has more than one "top" and is better described as providing a set of services.
- It is preferable not to pay too much attention to ordering constraints during the early stages of system analysis and design. Many temporal constraints can be described more abstractly as logical constraints.
- Top-down functional design is not appropriate for the long-term view of software systems, which involves change and reuse.
- Object-oriented software construction bases the structure of systems on the types of objects they manipulate.
- In object-oriented design, the primary design issue is not what the system does, but what types of objects it does it to. The design process defers to the last steps the decision as to what is the topmost function, if any, of the system.
- To satisfy the requirements of extendibility and reusability, object-oriented software construction needs to deduce the architecture from sufficiently abstract descriptions of objects.
- Two kinds of relation may exist between object types: client and inheritance.

5.7 BIBLIOGRAPHICAL NOTES

The case for object-based decomposition is made, using various arguments, in [Cox 1990] (original 1986), [Goldberg 1981], [Goldberg 1985], [Page-Jones 1995] and [M 1978], [M 1979], [M 1983], [M 1987], [M 1988].

The top-down method has been advocated in many books and articles. [Wirth 1971] developed the notion of stepwise refinement.

Of other methods whose rationales start with some of the same arguments that have led this discussion to object-oriented concepts, the closest is probably Jackson's JSD [Jackson 1983], a higher-level extension of JSP [Jackson 1975]. See also Warnier's data-directed design method [Orr 1977]. For a look at the methods that object technology is meant to replace, see books on: Constantine's and Yourdon's structured design [Yourdon 1979]; structured analysis [DeMarco 1978], [Page-Jones 1980], [McMenamin 1984], [Yourdon 1989]; Merise [Tardieu 1984], [Tabourier 1986].

Entity-relationship modeling was introduced by [Chen 1976].