22

How to find the classes

*F*oremost among the goals of object-oriented methodology, since the structure of O-O software is based on decomposition into classes, is that it should give us some advice on how to find these classes. Such is the purpose of the following pages. (In some of the literature you will see the problem referred to as "*finding the objects*", but by now we know better: what is at stake in our software architectures is not individual objects, but object types — classes.)

At first we should not expect too much. Finding classes is the central decision in building an object-oriented software system; as in any creative discipline, making such decisions right takes talent and experience, not to mention luck. Expecting to obtain infallible recipes for finding the classes is as unrealistic as would be, for an aspiring mathematician, expecting to obtain recipes for inventing interesting theories and proving their theorems. Although both activities — software construction and theory construction — can benefit from general advice and the example of successful predecessors, both also require creativity of the kind that cannot fully be covered by mechanical rules. If (like many people in the industry) you still find it hard to compare the software developer to a mathematician, just think of other forms of engineering design: although it is possible to provide basic guidelines, no teachable step-by-step rules can guarantee good design of buildings or airplanes.

In software too, no book advice can replace your know-how and ingenuity. The principal role of a methodological discussion is to indicate some good ideas, draw your attention to some illuminating precedents, and alert you to some known pitfalls.

This would be true with any other software design method. In the case of object technology, the observation is tempered by some good news, coming to us in the form of reuse. Because much of the necessary invention may already have been done, you can build on others' accomplishments.

There is more good news. By starting with humble expectations but studying carefully what works and also what does not, we will be able, little by little and against all odds, to devise what in the end deserves to be called a *method* for finding the classes. One of the key steps will be the realization that, as always in design, a selection technique is defined by two components: what to consider, and what to reject.

22.1 STUDYING A REQUIREMENTS DOCUMENT

To understand the problem of finding classes, it may be best to begin by assessing a widely publicized approach.

The nouns and the verbs

A number of publications suggest using a simple rule for obtaining the classes: start from the requirements document (assuming there is one, of course, but that is another story); in function-oriented design you would concentrate on the verbs, which correspond to actions ("do this"); in object-oriented design you underline the nouns, which describe objects. So according to this view a sentence of the form

See the bibliographical notes.

The elevator will close its door before it moves to another floor.

would lead the function-oriented designer to detect the need for a "move" function; but as an object-oriented designer you should see in it three object types, *ELEVATOR*, *DOOR* and *FLOOR*, which will give classes. Voilà!

Would it that life were that simple. You would bring your requirements documents home at night, and play *Object Pursuit* around the dinner table. A good way to keep the children away from the TV set, and make them revise their grammar lessons while they help Mom and Dad in their software engineering work.

But such a simple-minded technique cannot take us very far. Human language, used to express system requirements, is so open to nuance, personal variation and ambiguity that it is dangerous to make any important decision on the basis of a document which may be influenced as much by the author's individual style as by the actual properties of the projected software system.

Any useful result that the "underline the nouns" method would give us is obvious anyway. Any decent O-O design for an elevator control system will include an *ELEVATOR* class. Obtaining such classes is not the difficult part. To repeat an expression used in an earlier discussion, they are here for the picking. For the non-obvious classes a syntactic criterion — such as nouns versus verbs in a document that is by essence open to many possible stylistic variants — is close to useless.

Although by itself the "underline the nouns" idea would not deserve much more consideration, we can use it further, not for its own sake but as a foil; by understanding its limitations we can gain insights into what it truly takes to find the classes and how the requirements document can help us in this endeavor.

Avoiding useless classes

The nouns of a requirements document will cover some classes of the final design, but will also include many "false alarms": concepts that should *not* yield classes.

In the elevator example *door* was a noun. Do we need a class *DOOR*? Maybe, maybe not. It is possible that the only relevant property of elevator doors for this system is that

they may be opened and closed. Then to express the useful properties of doors it suffices to include in class *ELEVATOR* the query and commands

```
door_open: BOOLEAN;
close_door is
        …
        ensure
                not door_open
        end;
open_door is
        …
        ensure
                door_open
        end
```

In another variant of the system, however, the notion of door may be important enough to justify a separate class. The only resource here is the theory of abstract data types, and the only relevant question is:

> Is "door" a separate data type with its own clearly identified operations, or are all the operations on doors already covered by operations on other data types such as *ELEVATOR*?

Only your intuition and experience as a designer will tell you the answer. In looking for it, you will be aided by the requirements document, but do not expect grammatical criteria to be of more than superficial help. Turn instead to the ADT theory, which will help you ask customers or future users the right questions.

Chapter 21.

We encountered a similar case in the undo-redo mechanism design. The discussion distinguished between *commands*, such as the line insertion command in a text editor, and the more general notion of *operation*, which includes commands but also special requests such as Undo. Both of these words figured prominently in the statement of the problem; yet only *COMMAND* yielded a data abstraction (one of the principal classes of the design), whereas no class in the solution directly reflects the notion of operation. No analysis of a requirements document can suggest this striking difference of treatment.

Is a new class necessary?

Another example of a noun which may or may not give a class in the elevator example is *floor*. Here (as opposed to the *door* and *operation* cases) the question is not whether the concept is a relevant ADT: floors are definitely an important data abstraction for an elevator system. But this does not necessarily mean we should have a *FLOOR* class.

The reason is simply that the properties of floors may be entirely covered, for the purposes of the elevator system, by those of integers. Each floor has a floor number; then

if a floor (as seen by the elevator system) has no other features than those associated with its floor number, you may not need a separate *FLOOR* class. A typical floor feature that comes from a feature of integers is the distance between two floors, which is simply the difference of their floor numbers.

Many hotels have no floor 13, so the arithmetic may be a bit more elaborate.

If, however, floors have properties other than those of their numbers — that is to say, according to the principles of abstract data types and object-oriented software construction, significant *operations* not covered by those of integers — then a *FLOOR* class will be appropriate. For example, some floors may have special access rights defining who can visit them; then the *FLOOR* class could include a feature such as

rights: *SET* [*AUTHORIZATION*]

and the associated procedures. But even that is not certain: we might get away by including in some other class an array

floor_rights: *ARRAY* [*SET* [*AUTHORIZATION*]]

which simply associates a set of *AUTHORIZATION* values with each floor, identified by its number.

Another argument for having a specific class *FLOOR* would be to limit the available operations: it makes sense to subtract two floors and to compare them (through the *infix* "<" function), but not to add or multiply them. Such a class may be written as an heir to *INTEGER*. The designer must ask himself, however, whether this goal really justifies adding a new class.

See exercise E22.1, page 745.

This discussion brings us once again to the theory of abstract data types. A class does not just cover physical "objects" in the naïve sense. It describes an abstract data type — a set of software objects characterized by well-defined operations and formal properties of these operations. A type of real-world objects may or may not have a counterpart in the software in the form of a type of software objects — a class. When you are assessing whether a certain notion should yield a class or not, only the ADT view can provide the right criterion: do the objects of the system under discussion exhibit enough specific operations and properties of their own, relevant to the system and not covered by existing classes?

The qualification "relevant to the system" is crucial. The aim of systems analysis is not to "model the world". This may be a task for philosophers, but the builders of software systems could not care less, at least for their professional activity. The task of analysis is to model that part of the world which is meaningful for the software under study or construction. This principle is reinforced by the ADT approach (that is to say, the object-oriented method), which holds that *objects are only defined by what we can do with them* — what the discussion of abstract data types called the Principle of Selfishness. If an operation or property of an object is irrelevant to the purposes of the system, then it should not be included in the result of your analysis — however interesting it may be for other purposes. For a census processing system, the notion of *PERSON* may have features *mother* and *father*; but for a payroll processing system which does not require information about the parents, every *PERSON* is an orphan.

"BEYOND SOFTWARE", 6.6, page 147.

If all of the operations and properties that you can identify for a type of objects are irrelevant in this sense, or are already covered by the operations and properties of a previously identified class, the conclusion is that the object type itself is irrelevant: it must not yield a class.

This explains why an elevator system might not include *FLOOR* as a class because (as noted above) from the point of view of the elevator system floors have no relevant properties other than those of the associated integer numbers, whereas a Computer Aided Design system designed for architects will have a *FLOOR* class — since in that case the floor has several specific attributes and routines.

Missing important classes

Not only can nouns suggest notions which do not yield classes: they can also fail to suggest some notions which should definitely yield classes. There are at least three sources of such accidents.

> Do not forget that, as noted, the aim of this discussion is no longer to convince ourselves of the deficiencies of the "underline the nouns" approach, whose limitations are by now so obvious that the exercise would not be very productive. Instead, we are analyzing these limitations as a way to gain more insight into the process of discovering classes.

The first cause of missed classes is simply due to the flexibility and ambiguity of human language — the very qualities that make it suitable for an amazingly wide range of applications, from speeches and novels to love letters, but not very reliable as a medium for accurate technical documents. Assume the requirements document for our elevator example contains the sentence

> *A database record must be created every time the elevator moves from one floor to another.*

The presence of the noun "record" suggests a class *DATABASE_RECORD*; but we may totally miss a more important data abstraction: the notion of a *move* between two floors. With the above sentence in the requirements document, you will almost certainly need a *MOVE* class, which could be of the form

```
class MOVE feature
        initial, final: FLOOR;        -- Or INTEGER if no FLOOR class
        record (d: DATABASE) is …
        … Other features …
end -- class MOVE
```

This will be an important class, which a grammar-based method would miss because of the phrasing of the above sentence. Of course if the sentence had appeared as

> *A database record must be created for every move of the elevator from one floor to another.*

then "move" would have been counted as a noun, and so would have yielded a class! We see once again the dangers of putting too much trust in a natural-language document, and the absurdity of making any serious property of a system design, especially its modular structure, dependent on such vagaries of style and mood.

The second reason for overlooking classes is that some crucial abstractions may not be directly deducible from the requirements. Cases abound in the examples of this book. It is quite possible that the requirements for a panel-driven system did not explicitly cite the notions of state and application; yet these are the key abstractions, which condition the entire design. It was pointed out earlier that some external-world object types may have no counterpart among the classes of the software; here we see the converse: classes of the software that do not correspond to any external-world objects. Similarly, if the author of the requirements for a text editor with undo-redo has written "*the system must support line insertion and deletion*", we are in luck since we can spot the nouns *insertion* and *deletion*; but the need for these facilities may just as well follow from a sentence of the form

Panel-driven system: *chapter 20. Undo-redo*: *chapter 21*.

> *The editor must allow its users to insert or delete a line at the current cursor position.*

leading the naïve designer to devote his attention to the trivial notions of "cursor" and "position" while missing the command abstractions (line insertion and line deletion).

The third major cause of missed classes, shared by any method which uses the requirements document as the basis for analysis, is that such a strategy overlooks reuse. It is surprising to note that much of the object-oriented analysis literature takes for granted the traditional view of software development: starting from a requirements document and devising a solution to the specific problem that it describes. One of the major lessons of object technology is the lack of a clear-cut distinction between problem and solution. Existing software can and should influence new developments.

See "THE CHANGING NATURE OF ANALYSIS", 27.2, page 906.

When faced with a new software project, the object-oriented software developer does not accept the requirements document as the alpha and omega of wisdom about the problem, but combines it with knowledge about previous developments and available software libraries. If necessary, he will criticize the requirements document and propose updates and adaptations which will facilitate the construction of the system; sometimes a minor change, or the removal of a facility which is of limited interest to the final users, will produce a dramatic simplification by making it possible to reuse an entire body of existing software and, as a result, to decrease the development time by months. The corresponding abstractions are most likely to be found in the existing software, not in the requirements document for the new project.

Classes *COMMAND* and *HISTORY_LOG* from the undo-redo example are typical. The way to find the right abstractions for this problem is not to rack one's brain over the requirements document for a text editor: either you come upon them through a process of intellectual discovery (a "Eureka", for which no sure recipe exists); or, if someone else has already found the solution, you reuse his abstractions. You may of course be able to reuse the corresponding implementation too if it is available as part of a library; this is even better, as the whole analysis-design-implementation work has already been done for you.

Discovery and rejection

> *It takes two to invent anything. One makes up combinations; the other chooses, recognizes what is important to him in the mass of things which the first has imparted to him. What we call genius is much less the work of the first than the readiness of the second to choose from what has been laid before him.*
>
> Paul Valéry (cited in [Hadamard 1945]).

Along with its straightforward lessons, this discussion has taught us a few more subtle consequences.

The simple lessons have been encountered several times: do not put too much trust in a requirements document; do not put *any* trust in grammatical criteria.

A less obvious lesson has emerged from the review of "false alarms": just as we need criteria for finding classes, we need criteria for **rejecting** candidate classes — concepts which initially appear promising but end up not justifying a class of their own. The design discussions of this book illustrate many such cases.

"Pseudo-random number generators: a design exercise", page 754.

> To quote just one example: a discussion, yet to come, of how best to provide for pseudo-random number generation, starts naturally enough by considering the notion of random number, only to dismiss it as not the appropriate data abstraction.

The O-O analysis and design books that I have read include little discussion of this task. This is surprising because in the practice of advising O-O projects, especially with relatively novice teams, I have found that eliminating bad ideas is just as important as finding good ones.

It may even be more important. Sit down with a group of users, developers and managers trying to get started with object technology with a fresh new project and enthusiasm fresher yet. There will be no dearth of ideas for classes (usually proposed as "objects"). The problem is to dam the torrent before it damns the project. Although some class ideas will probably have been missed, many more will have to be examined and rejected. As in a large-scale police investigation, many leads come in, prompted or spontaneous; you must sort the useful ones from the canards.

So we must adapt and extend the question that serves as the topic for this chapter. "How to find the classes" means two things: not just how to come up with candidate abstractions but also how to unmask the inadequate among them. These two tasks are not executed one after the other; instead, they are constantly interleaved. Like a gardener, the object-oriented designer must all the time nurture the good plants and weed out the bad:

Class Elicitation principle

Class elicitation is a dual process: class suggestion, class rejection.

The rest of this chapter studies both components of the class elicitation process.

22.2 DANGER SIGNALS

To guide our search it is preferable to start with the rejection part. It will provide us with a checklist of typical pitfalls, alert us to the most important criteria, and help us keep our search for good classes focused on the most productive efforts.

Let us review a few signs that usually indicate a bad choice of class. Because design is not a completely formalized discipline, you should not treat these signs as *proof* of a bad design; in each case one can think of some circumstances that may make the original decision legitimate. So what we will see is not, in the terms of a previous chapter, "absolute negatives" (sure-fire rules for rejecting a design) but "advisory negatives": danger signals that alert you to the presence of a suspicious pattern, and should prompt you to investigate further. Although in most cases they should lead you to revise the design, you may occasionally decide in the end that it is right as it stands.

"A typology of rules", page 666.

The grand mistake

Many of the danger signals discussed below point to the most common and most damaging mistake, which is also the most obvious: designing a class that isn't.

The principle of object-oriented software construction is to build modules around object types, not functions. This is the key to the reusability and extendibility benefits of the approach. But beginners will often fall into the most obvious pitfall: calling "class" something which is in fact a routine. Writing a module as **class**… **feature** … **end** does not make it a true class; it may just be a routine in disguise.

This Grand Mistake is easy to avoid once you are conscious of the risk. The remedy is the usual one: make sure that each class corresponds to a meaningful data abstraction.

What follows is a set of typical traits alerting you to the risk that a module which presents itself as a candidate class, and has the syntactical trappings of a class, may be an illegal immigrant not deserving to be granted citizenship in the O-O society of modules.

My class performs…

In a design meeting, an architecture review, or simply an informal discussion with a developer, you ask about the role of a certain class. The answer: "*This class prints the results*" or "*this class parses the input*", or some other variant of "*This class does…*".

The answer usually points to a design flaw. A class is not supposed to *do* one thing but to offer a number of services (features) on objects of a certain type. If it really does just one thing, it is probably a case of the Grand Mistake: devising a class for what should just be a routine of some other class.

Perhaps the mistake is not in the class itself but in the way it is being described, using phraseology that is too operational. But you had better check.

> In recent years the "*my class does…*" style has become widespread. A NeXT document describes classes as follows: "*The NSTextView class declares the programmatic interface to objects that display text laid out…*"; "*An NSLayoutManager coordinates the layout*

NeXT documentation for OpenStep, pre-release 4.0.

and display of characters…"; "*NSTextStorage is a semi-concrete subclass of NSMutableAttributedString that manages a set of client NSLayoutManagers, notifying them of any changes…*". Even if (as is most likely the case here) the classes discussed represent valuable data abstractions, it would be preferable to describe them less operationally by emphasizing these abstractions.

Imperative names

Assume that in a tentative design you find a class name such as *PARSE* or *PRINT* — a verb in the imperative or infinitive. It should catch your attention, as signaling again a probable case of a class that "does one thing", and should not be a class.

Occasionally you may find that the class is right. Then its name is wrong. This is an "absolute positive" rule:

Class Name rule

A class name must always be either:

- A noun, possibly qualified.
- (Only for a deferred class describing a structural property) an adjective.

Although like any other one pertaining to style this rule is partly a matter of convention, it helps enforce the principle that every class represents a data abstraction

The first form, nouns, covers the vast majority of cases. A noun may be used by itself, as in *TREE*, or with some qualifying words, as in *LINKED_LIST*, qualified by an adjective, and *LINE_DELETION*, qualified by another noun.

"Structure inheritance", page 831.

The second case, adjectives, arises only for a specific case: *structural property* classes describing an abstract structural property, as with the Kernel Library class *COMPARABLE* describing objects on which a certain order relation is available. Such classes should be deferred; their names (in English or French) will often end with *ABLE*. They are meant to be used through inheritance to indicate that all instances of a class have a certain property; for example in a system for keeping track of tennis rankings class *PLAYER* might inherit from *COMPARABLE*. In the taxonomy of inheritance kinds, this scheme will be classified as *structure inheritance*.

See chapter 21.

The only case that may seem to suggest an exception to the rule is command classes, as introduced in the undo-redo design pattern to cover action abstractions. But even then you should stick to the rule: call a text editor's command classes *LINE_DELETION* and *WORD_CHANGE*, not *DELETE_LINE* and *REPLACE_WORD*.

English leaves you more flexibility in the application of this rule than many other languages, since its grammatical categories are more an article of faith than an observation of fact, and almost every verb can be nouned. If you use English as the basis for the names in your software it is fair to take advantage of this flexibility to devise shorter and simpler names: you may call a class *IMPORT* where other languages might treat the equivalent as a verb only, forcing you to use nouns such as *IMPORTATION*. But do not cheat: class

IMPORT should cover the abstraction "objects being imported" (nominal), not, except for a command class, the act of importing (verbal).

> It is interesting to contrast the Class Name rule with the discussion of the "underline the nouns" advice at the beginning of this chapter. "Underline the nouns" applied a formal grammatical criterion to an informal natural-language text, the requirements document; this is bound to be of dubious value. The Class Name rule, on the other hand, applies the same criterion to a *formal* text — the software.

Single-routine classes

A typical symptom of the Grand Mistake is an effective class that contains only one exported routine, possibly calling a few non-exported ones. The class is probably just a glorified subroutine — a unit of functional rather than object-oriented decomposition.

A possible exception arises for objects that legitimately represent abstracted actions, for example a command in an interactive system, or what in a non-O-O approach would have been represented by a routine passed as argument to another routine. But the examples given in an earlier discussion show clearly enough that even in such cases there will usually be several applicable features. We noted that a mathematical software object representing a function to be integrated will not just have the feature *item* (*a*: *REAL*): *REAL*, giving the value of the function at point *a*: others may include domain of definition, minimum and maximum over a certain interval, derivative. Even if a class does not yet have all these features, checking that it would make sense to add them later will reinforce your conviction that you are dealing with a genuine object abstraction.

See "Small classes", page 714.

In applying the single-routine rule, you should consider all the features of a class: those introduced in the class itself, and those which it inherits from its parents. It is not necessarily wrong for a class text to declare only one exported routine, if this is simply an addition to a meaningful abstraction defined by its ancestors. It may, however, point to a case of *taxomania*, an inheritance-related disease which will be studied as part of the methodology of inheritance.

See "TAXOMANIA", 24.4, page 820.

Premature classification

The mention of taxomania suggests a warning about another common mistake of novices: starting to worry about the inheritance hierarchy too early in the process.

As inheritance is central in the object-oriented method, so is a good inheritance structure — more accurately, a good modular structure, including both inheritance and client relations — essential to the quality of a design. But inheritance is only relevant as a relation among well-understood abstractions. When you are still looking for the abstractions, it is too early to devise the inheritance hierarchy.

The only clear exception arises when you are dealing with an application domain for which a pre-existing taxonomy is widely accepted, as in some branches of science. Then the corresponding abstractions will emerge together with their inheritance structure. (Before accepting the taxonomy as the basis for your software's structure, do check that it is indeed well recognized and stable, not just someone's view of things.)

In other cases, you should only design the inheritance hierarchy once you have at least a first grasp of the abstractions. (The classification effort may of course lead you to revise your choice of abstractions, prompting an iterative process in which the tasks of class elicitation and inheritance structure design feed each other.) If, early in a design process, you find the participants focusing on classification issues even though the classes are not yet well understood, they are probably putting the cart before the horse.

With novices, this may be a variant of the object-class confusion. I have seen people start off with inheritance hierarchies of the "*SAN_FRANCISCO* and *HOUSTON* inherit from *CITY*" kind — simply to model a situation where a single class, *CITY*, will have several instances at run time.

No-command classes

Sometimes you will find a class that has no routine at all, or only provides queries (ways to access objects) but no commands (procedures to modify objects). Such a class is the equivalent of a record in Pascal or a structure in Cobol or C. It may indicate a design mistake, but the mistake may be of two kinds and you will need to probe further.

First, let us examine three cases in which the class does *not* indicate improper design:

- It may represent objects obtained from the outside world, which the object-oriented software cannot change. They could be data coming from a sensor in a process-control system, packets from a packet-switching network, or C structures that the O-O system is not supposed to touch.
- Some classes are meant not for direct instantiation, but for encapsulating facilities such as constants, used by other classes through inheritance. Such *facility inheritance* will be studied in the discussion of inheritance methodology.

"FACILITY INHERITANCE", 24.9, page 847.

- Finally, a class may be *applicative*, that is to say describe non-modifiable objects; instead of commands to modify an object it will provide functions that produce new objects, usually of the same type. For example the addition operation in classes *INTEGER*, *REAL* and *DOUBLE* follows the lead of mathematics: it does not modify any value but, given two values *x* and *y*, produces a third one *x* + *y*. In the abstract data type specification such functions will, like others that yield commands, be characterized as command functions.

Command functions were defined in "Function categories", page 134.

In all these cases the abstractions are easy to recognize, so you should have no difficulty identifying the two cases that may indeed point to a design deficiency.

Now for these suspicious cases. In the first one, the class is justified and would need commands; the designer has simply forgotten to provide mechanisms to modify the corresponding objects. A simple checklist technique presented in the discussion of class design will help avoid such mistakes.

See "A checklist", page 770.

In the second case, most directly relevant to this discussion, the class was not justified. It is not a real data abstraction, simply some piece of passive information which might have been represented by a structure such as a list or array, or just by adding more attributes to another class. This case sometimes happens when developers write a class for

what would have been a simple record (structure) type in Pascal, Ada or C. Not all record types cover separate data abstractions.

You should investigate such a case carefully to try to understand whether there is room for a legitimate class, now or in the future. If the answer is unclear, you may be better off keeping the class anyway even if it risks being overkill. Having a class may imply some performance overhead if it means dealing with many small objects, dynamically created one by one and occupying more space than simple array elements; but if you do need a class and have not introduced it early enough, the adaptation may take some effort.

> We had such a false start in the history of ISE's compiler. A compiler for an O-O language needs some internal way to identify each class of a system it processes; the identification used to be an integer. This worked fine for several years, but at some point we needed a more elaborate class identification scheme, allowing us in particular to *renumber* classes when merging several systems. The solution was to introduce a class *CLASS_IDENTIFIER*, and to replace the earlier integers by instances of that class. The conversion effort was more than we would have liked, as usually happens when you have missed an important abstraction. Initially *INTEGER* was a sufficient abstraction because no commands were applicable to class identifiers; the need for more advanced features, in particular renumbering commands, led to the recognition of a separate abstraction.

Mixed abstractions

Another sign of an imperfect design is a class whose features relate to more than one abstraction.

> In an early release of the NeXT library, the text class also provided full visual text editing capabilities. Users complained that the class, although useful, was too big. Large class size was the symptom; the true problem was the merging of two abstractions (character string, and interactively editable text); the solution was to separate the two abstractions, with a class *NSAttributedString* defining the basic string handling mechanism and various others, such as *NSTextView*, taking care of the user interface aspects.

Meilir Page-Jones uses the term *connascence* (defined in dictionaries as the property of being born and having grown together) to describe the relation that exists between two features when they are closely connected, based on a criterion of simultaneous change: a change to one will imply a change to the other. As he points out, you should minimize connascence across class libraries; but features that appear within a given class should all be related to the same clearly identified abstraction. [Page-Jones 1995].

This universal guideline deserves to be expressed as a methodological rule (presented in "positive" form although it follows a discussion of possible mistakes):

Class Consistency principle

All the features of a class must pertain to a single, well-identified abstraction.

The ideal class

This review of possible mistakes highlights, by contrast, what the ideal class will look like. Here are some of the typical properties:

- There is a clearly associated abstraction, which can be described as a data abstraction (or as an abstract machine).
- The class name is a noun or adjective, adequately characterizing the abstraction.
- The class represents a set of possible run-time objects, its instances. (Some classes are meant to have only one instance during an execution; that is acceptable too.)
- Several queries are available to find out properties of an instance.
- Several commands are available to change the state of an instance. (In some cases, there are no commands but instead functions producing other objects of the same type, as with the operations on integers; that is acceptable too.)
- Abstract properties can be stated, informally or (preferably) formally, describing: how the results of the various queries relate to each other (this will yield the invariant); under what conditions features are applicable (preconditions); how command execution affects query results (postconditions).

This list describes a set of informal goals, not a strict rule. A legitimate class may have only some of the properties listed. Most of the examples that play an important role in this book — from *LIST* and *QUEUE* to *BUFFER*, *ACCOUNT*, *COMMAND*, *STATE*, *INTEGER*, *FIGURE*, *POLYGON* and many others — have them all.

22.3 GENERAL HEURISTICS FOR FINDING CLASSES

Let us now turn to the positive part of our discussion: practical heuristics for finding classes.

Class categories

We may first note that there are three broad categories of classes: analysis classes, design classes and implementation classes. The division is neither absolute nor rigorous (for example one could find arguments to support attaching a deferred class *LIST* to any one of the three categories), but it is convenient as a general guideline.

An analysis class describes a data abstraction directly drawn from the model of the external system. *PLANE* in a traffic control system, *PARAGRAPH* in a document processing system, *PART* in an inventory control system are typical examples.

An implementation class describes a data abstraction introduced for the internal needs of the algorithms in the software, such as *LINKED_LIST* or *ARRAY*.

In-between, a design class describes an architectural choice. Examples included *COMMAND* in the solution to the undo-redo problem, and *STATE* in the solution to the problem of panel-driven systems. Like implementation classes, design classes belong to the *solution* space, whereas analysis classes belong to the problem space. But like analysis classes and unlike implementation classes they describe high-level concepts.

As we study how to obtain classes in these three categories, we will find that design classes are the most difficult to identify, because they require the kind of architectural

insight that sets the gifted designer apart. (That they are the most difficult to find does not mean they are the most difficult to *build*, a distinction that usually belongs to the implementation classes, unless of course you come across a ready-to-be-reused implementation library.)

External objects: finding the analysis classes

Let us start with the analysis classes, modeled after external objects.

We use software to obtain answers to certain questions about the world (as in a program that computes the solution to a specific problem), to interact with the world (as in a process control system), or to add things to the world (as in a text processing system). In every case, the software must be based on some model of the aspects of the world that are relevant to the application, such as laws of physics or biology in a scientific program, the syntax and semantics of a computer language in a compiler, salary scales in a payroll system, and income tax regulations in tax processing software.

To talk about the world being modeled we should avoid the term "real world", which is misleading, both because software is no less "real" than anything else and because many of the non-software "worlds" of interest are artificial, as in the case of a mathematical program dealing with equations and graphs. (An earlier chapter discussed this question in detail.) We should talk about the *external world*, as distinct from the internal world of the software that deals with it.

See "Reality: a cousin twice removed", page 230.

Any software system is based on an **operational model** of some aspect of the external world. Operational because it is used to generate practical results and sometimes to feed these results back into the world; model because any useful system must follow from a certain interpretation of some world phenomena.

Nowhere perhaps is this view of software as inescapable as in the area of *simulation*. It is no accident that the first object-oriented language, Simula 67, evolved from Simula 1, a language for writing discrete-event simulations. Although Simula 67 itself is a general-purpose programming language, it retained the name of its predecessor and includes a set of powerful simulation primitives. Well into the nineteen-seventies, simulation remained the principal application area of object technology (as a look into the proceedings of the annual Association of Simula Users conferences suffices to show). This attraction of O-O ideas for simulation is easy to understand: to devise the structure of a software system simulating the behavior of a set of external objects, what could be better than using software components which directly represent those objects?

See "SIMULA", 35.1, page 1113.

In a broad sense, of course, all software is simulation. Capitalizing on this view of software as operational modeling, object-oriented software construction uses as its first abstractions some types deduced from analyzing the principal types of objects, in the non-software sense of the term, in the external world: sensors, devices, airplanes, employees, paychecks, tax returns, paragraphs, integrable functions.

These examples, by the way, suggest only part of the picture. As Waldén and Nerson note in their presentation of the B.O.N. method:

[Waldén 1995], *pages 182-183*.

> *A class representing a car is no more tangible than one that models the job satisfaction of employees. What counts is how important the concepts are to the enterprise, and what you can do with them.*

Keep this comment in mind when looking for external classes: they can be quite abstract. *SENIORITY_RULE* for a parliament voting system and *MARKET_TENDENCY* for a trading system may be just as real as *SENATOR* and *STOCK_EXCHANGE*. The smile of the Cheshire Cat has as much claim to objectness as the Cheshire Cat.

Whether material or abstract, external classes represent the abstractions that specialists of the external world, be they aerospace engineers, accountants or mathematicians, constantly use to think and talk about their domain. There is always a good chance — although not a certainty — that such an object type will yield a useful class, because typically the domain experts will have associated significant operations and properties with it.

The key word, as usual, is *abstraction*. Although it is desirable that analysis classes closely match concepts from the problem domain, this is not what makes a candidate class good. The first version of our panel-driven system dramatically showed why: there we had a model directly patterned after some properties of the external system, but terrible from a software engineering viewpoint because the selected properties were low-level and subject to change. A good external class will be based on abstract concepts of the problem domain, characterized (in the ADT way) through external features chosen because of their lasting value.

For the object-oriented developer such pre-existing abstractions are precious: they provide some of the system's fundamental classes; and, as we may note once more, the objects are here for the picking.

Finding the implementation classes

Implementation classes describe the structures that software developers use to make their systems run on a computer. Although the fashion in the software engineering literature has been, for the past fifteen years, to downplay the role of implementation, developers know the obvious — that implementation consumes a large part of the effort in building a system, and much of the intelligence that goes into it.

The bad news is that implementation is difficult. The good news is that implementation classes, although often hard to *build* in the absence of good reusable libraries, are not the most difficult to *elicit*, thanks to the ample body of literature on the topic. Since "Data Structures and Algorithms", sometimes known as "CS 2", is a required component of computing science education, many textbooks survey the rich catalog of useful data structures that have been identified over the years. Better yet, although most existing textbooks do not explicitly use an object-oriented approach, many naturally follow an abstract data type style, even if they do not use the phrase, to present data structures; for example to introduce various forms of table such as binary search trees and hash tables you have first to state the various operations (insert an element with its key, search for an element through its key and so on) with their properties. The transition to classes is fairly straightforward.

Recently, some textbooks have started to go further by applying a thoroughly object-oriented approach to the traditional CS 2 topics.

Whether or not he has gone through a Data Structures and Algorithms Course at school, every software engineer should keep a good textbook on the topic within reach of hand, and go back to it often. It is all too easy to waste time reinventing concepts that are well known, implement a less-than-optimal algorithm, or choose a representation that is not appropriate for the software's use of a data structure — for example a one-way linked list for a sequential structure that the algorithms must regularly traverse back and forth, or an array for a structure that constantly grows and shrinks in unpredictable ways. Note that here too the ADT approach reigns: the data structure and its representation follow from the services offered to clients.

Beyond textbooks and experience, the best hope for implementation classes is reusable libraries, as we will see at the end of this chapter.

Deferred implementation classes

Traditional data structures textbooks naturally emphasize effective (fully implemented) classes. In practice, much of the value of a set of implementation classes, especially if they are meant to be reusable, lies in the underlying taxonomy, as defined by an inheritance structure that will include deferred classes. For example, various queue implementations will be descendants of a deferred class *QUEUE* describing the abstract concept of sequential list.

"Deferred implementation class", then, is not an oxymoron. Classes such as *QUEUE*, although quite abstract, help build the taxonomies thanks to which we can keep the many varieties of implementation structures coherent and organized, assigning to every class a precise place in the overall scheme.

In another book [M 1994a] I have described a "Linnaean" taxonomy of the fundamental structures of computing science, which relies on deferred classes to classify the principal kinds of data structure used in software development.

Finding the design classes

Design classes represent architectural abstractions that help produce elegant, extendible software structures. *STATE*, *APPLICATION*, *COMMAND*, *HISTORY_LIST*, iterator classes, "controller" classes as in the Smalltalk MVC model are good examples of design classes. We will see other seminal ideas in subsequent chapters, such as active data structures and "handles" for platform-adaptable portable libraries.

About iterators and MVC see the bibliographical notes.

Although, as noted, there is no sure way to find design classes, a few guidelines are worth noting:

- Many design classes have been devised by others before. By reading books and articles that describe precise solutions to design problems, you will gain many fruitful ideas. For example the book *Object-Oriented Applications* contains chapters written by the lead designers of various industrial projects who describe their

[M 1993].

architectural solutions in detail, providing precious guidance to others faced with similar problems in telecommunications, Computer-Aided Design, artificial intelligence and other application areas.

[Gamma 1995].

- The book on "design patterns" by Gamma *et al.* has started an effort of capturing proven design solutions and is now being followed by several others.
- Many useful design classes describe abstractions that are better understood as machines than as "objects" in the common (non-software) sense.
- As with implementation classes, reuse is preferable to invention. One can hope that many of the "patterns" currently being studied will soon cease to be mere ideas, yielding instead directly usable library classes.

22.4 OTHER SOURCES OF CLASSES

A number of heuristics have proved useful in the quest for the right abstractions.

Previous developments

The advice of looking first at what is available does not just apply to library classes. As you write applications, you will accumulate classes which, if properly designed, should facilitate later developments.

See "GENERALIZATION", 28.5, page 928.

Not all reusable software was born reusable. Often, the first version of a class is produced to meet some immediate requirement rather than for posterity. If reusability is a concern, however, it pays to devote some time, after the development, to making the class more general and robust, improving its documentation, adding assertions. This is different from the construction of software meant from the start to be reusable, but no less fruitful. Having evolved from components of actual systems, the resulting classes have passed the first test of reusability, namely *usability*: they serve at least one useful purpose.

Adaptation through inheritance

When you discover the existence of a potentially useful class, you will sometimes find that it does not exactly suit your present need: some adaptation may be necessary.

Unless the adaptation addresses a deficiency which should be corrected in the original as well, it is generally preferable to leave the class undisturbed, preserving its clients according to the Open-Closed principle. Instead, you may use inheritance and redefinition to tune the class to your new need.

See "Variation inheritance", page 828.

This technique, which our later taxonomy of uses of inheritance will study in detail under the name *variation inheritance*, assumes that the new class describes a variant of the same abstraction as the original. If used properly (according to the guidelines of the later discussion) it is one of the most remarkable contributions of the method, enabling you to resolve the *reuse-redo* dilemma: combining reusability with extendibility.

Evaluating candidate decompositions

Criticism is said to be easier than art; a good way to learn design is to learn to analyze existing designs. In particular, when a certain set of classes has been proposed to solve a certain problem, you should study them from the criteria and principles of modularity given in chapter 3: do they constitute autonomous, coherent modules, with strictly controlled communication channels? Often, the discovery that two modules are too tightly coupled, that a module communicates with too many others, that an argument list is too long, will pinpoint design errors and lead to a better solution.

An important criterion was explored in the panel-driven system example: data flow. We saw then how important it is to study, in a candidate class structure, the flow of objects passed as arguments in successive calls. If, as with the notion of State in that example, you detect that a certain item of information is transmitted over many modules, it is almost certainly a sign that you have missed an important data abstraction. Such an analysis, which we applied to obtain the class *STATE*, is an important source of abstractions.

Chapter 20.

It is of course preferable to find the classes right from the start; but better late than never. After such an a posteriori class discovery, you should take the time to analyze why the abstraction was initially missed, and to reflect on how to do better next time.

Hints from other approaches

The example of analyzing data flow in a top-down structure illustrates the general idea of deriving class insights from concepts of non-O-O decompositions. This will be useful in two non-disjoint cases:

- There may already exist a non-O-O software system which does part of the job; it may be interesting to examine it for class ideas. The same would apply if, instead of a working system, you can use the result of an analysis or design produced with another, older method.
- Some of the people doing the development may have had extensive experience with other methods, and as a consequence may initially think in terms of different concepts, some of which may be turned into class ideas.

Here are examples of this process, starting with programming languages and continuing with analysis and design techniques.

Fortran programs usually include one or more *common blocks* — data areas that can be shared by several routines. A common block often hides one or more valuable data abstractions. More precisely, good Fortran programmers know that a common block should only include a few variables or arrays, covering closely related concepts; there is a good chance that such a block will correspond to one class. Unfortunately, this is not universal practice, and even programmers who know better than to use the "garbage common block" mentioned at the beginning of this book tend to put too many things in one common block. In this case you will have to examine patterns of use of each block to discover the abstraction or abstractions that it covers.

On garbage common blocks see "Small Interfaces", page 48.

Pascal and C programs use records, known in C as structures. (Pascal only has record *types*; in C you can have structure types as well as individual structures.) A record type often corresponds to a class, but only if you can find operations acting specifically on instances of the type, usually (as we saw) including commands as well as queries. If not, the type may just represent some attributes of another class.

Cobol also has structures, and its Data Division helps identify important data types.

In entity-relationship (ER) modeling, analysts isolate "entities" which can often serve as seeds for classes.

> People with a long practice of ER modeling are among those who sometimes find it initially hard to apply object-oriented ideas effecively, because they are used to treating the entities and relationships as being different in nature, and the "dynamic" behavior of the system as completely separate from them. With O-O modeling both the relationships and the behavior yield features attached to the types of objects (entities); thinking of relations and operations as variants of the same notion, and attaching them to entities, sometimes proves to be a little hard to swallow at first.

In dataflow design ("structured analysis and design") there is little that can be directly used for an object-oriented decomposition, but sometimes the "stores" (database or file abstractions) can suggest an abstraction.

Files

The comment about stores suggests a more general idea, useful again if you are coming from a non-O-O background. Sometimes much of the intelligence of a traditional system is to be found outside of the software's text, in the structure of the files that it manipulates.

To anyone with Unix experience, this idea will be clear: for some of the essential information that you need to learn, the essential documentation is the description not of specific commands but of certain key files and their formats: *passwd* for passwords, *printcap* for printer properties, *termcap* or *terminfo* for terminal properties. One could characterize these files as data abstractions without the abstraction: although documented at a very concrete level ("*Each entry in the printcap file describes a printer, and is a line consisting of a number of fields separated by* **:** *characters. The first entry for each printer gives the names which are known for the printer, separated by* **|** *characters*", etc.), they describe important data types accessible through well-defined primitives, with some associated properties and usage conditions. In the transition to an object-oriented view, such files would play a central role.

A similar observation applies to many programs, whose principal files embody some of the principal abstractions.

I once participated in a consulting session with the manager of a software system who was convinced that the system — a collection of Fortran programs — could not lend itself to object-oriented decomposition. As he was describing what the programs did, he casually mentioned a few files through which the programs communicated. I started asking questions about these files, but initially he kept dismissing these questions as unimportant, immediately coming back to the programs. I insisted, and from his

explanations realized that the files described complex data structures embodying the programs' essential information. The lesson was clear: as soon as the relevance of these files was recognized, they conquered the central place in the object-oriented architecture; in an upheaval typical of object-oriented rearchitecturing, the programs, formerly the key elements of the architecture, became mere features of the resulting classes.

Use cases

Ivar Jacobson has advocated relying on use cases as a way to elicit classes. A use case, called a *scenario* by some other analysis and design authors (and a *trace* in theoretical computing science, especially the study of concurrency), is a description of

> *a complete course of events initiated by a* [user of the future system] *and* [of] *the interaction between* [the user] *and the system*.

[Jacobson 1992], *page 154. Jacobson uses the term "actor" for users of the future system.*

In a telephone switching system, for example, the use case "customer-initiated call" has the sequence of events: customer picks handset, identification gets sent to the system, system sends dial tone, and so on. Other use cases for the system might include "caller-id service installation" and "customer disconnection".

Use cases are a not a good tool for finding classes. Relying on them in any significant way raises several risks:

- Use cases emphasize ordering ("*When a customer places an order over the phone, his credit card number is validated. Then the database is updated and a confirmation number is issued*", etc.). This is incompatible with object technology: the method shuns early reliance on sequentiality properties, because they are so fragile and subject to change. The competent O-O analyst and designer refuses to focus on properties of the form "The system does *a*, then *b*"; instead, he asks the question "What are the operations available on instances of abstraction *A*, and the constraints on these operations?". The truly fundamental sequentiality properties will emerge in the form of high-level constraints on the operations; for example, instead of saying that a stack supports alternating sequences of *push* and *pop* operations with never more *pop* than *push*, we define the preconditions attached with each of these operations, which imply the ordering property but are more abstract. Less fundamental ordering requirements simply have no place in the analysis model as they destroy the system's adaptability and hence its future survival. Early emphasis on ordering is among the worst mistakes an O-O project can make. If you rely on use cases for analysis, this mistake is hard to avoid.

See "Ordering and O-O development", page 111 and "Structure and order: the software developer as arsonist", page 201.

- Relying on a scenario means that you focus on how users see the system's operation. But the system does not exist yet. (A previous system might exist, but if it were fully satisfactory you would not be asked to change or rewrite it.) So the system picture that use cases will give you is based on existing processes, computerized or not. Your task as a system builder is to come up with *new*, better scenarios, not to perpetuate antiquated modes of operation. There are enough examples around of computer systems that slavishly mimic obsolete procedures.

- Use cases favor a functional approach, based on processes (actions). This approach is the reverse of O-O decomposition, which focuses on data abstractions; it carries a serious risk of reverting, under the heading of object-oriented development, to the most traditional forms of functional design. True, you may rely on several scenarios rather than just one main program. But this is still an approach that considers *what the system does* as the starting point, whereas object technology considers *what it does it to*. The clash is irreconcilable.

The practical consequences are obvious. A number of teams that have embraced use cases find themselves, without realizing it, practicing top-down functional design ("*the system must do a, then b, …*") and building systems that are obsolete on the day they are released, yet hard to change because they are tied to a specific view of what the system does. I have sat, as an outside consultant, in design reviews for such projects, trying to push for more abstraction. But it is difficult to help, because the designers are convinced that they are doing object-oriented design; they expect the consultant to make a few suggestions, criticize a few details and give his blessing to the overall result. The designs that I saw were not object-oriented at all, and were bound to yield flawed systems; but trying to convey this observation politely was about as effective as telling the group that the sun was not shining outside — we work from use cases, and doesn't everyone know that use cases are O-O?

The risks are perhaps less severe with a very experienced object-oriented design team — experience being evidenced by the team's previous production of large and successful O-O systems, in the thousands of classes and hundreds of thousands of lines. Such a group might find use cases useful as a complement to other analysis techniques. But for a novice team, or one with moderate experience only, the benefits of use cases as an analysis tool are so uncertain, and the risk of destroying the quality of the future system so great, as to recommend staying away altogether from this technique:

Use Case principle

Except with a very experienced design team (having built several successful systems of several thousand classes each in a pure O-O language), do not rely on use cases as a tool for object-oriented analysis and design.

This principle does not mean that use cases are a worthless concept. They remain a potentially valuable tool but their role in object-oriented software construction has been misunderstood. Rather than an analysis tool they are a *validation* tool. If (as you should) you have a separate quality assurance team, it may find use cases useful as a way to inspect a proposed analysis model or tentative design for possibly missing features. The QA team can check that the system will be able to run the typical scenarios identified by the users. (In some cases of negative answer you may find that the model will support a different scenario that achieves the same or better results. This is of course satisfactory.)

Another possible application of use cases is to the final aspects of implementation, to make sure that the system includes routines for typical usage scenarios. Such routines will often be of the abstract behavior kind, describing a general effective scheme relying on deferred routines which various components of the system, and future additions to it, may redefine in different ways. ([Jacobson 1992] indeed mentions a notion of *abstract use case* that mirrors the object-oriented concept of behavior class.

In these two roles as a validation mechanism and an implementation guide, use cases can be beneficial. But in object technology they are not a useful analysis or design mechanism. The system analysts and builders should concentrate on the abstractions, not on particular ways of scheduling operations on these abstractions.

CRC cards

For completeness it is necessary to mention an idea that is sometimes quoted as a technique to find classes. CRC cards (*Class*, *Responsibility*, *Collaboration*) are paper cards, 4 inches by 6 inches (10.16 centimeters by 15.24 centimeters), on which designers discuss potential classes in terms of their responsibilities and how they communicate. The idea has the advantage of being easy on the equipment budget (a box of cards is typically cheaper than a workstation with CASE tools) and of fostering team interaction. Its technical contribution to the design process — to helping sort out and characterize valuable abstractions — is, however, unclear.

K. Beck and W. Cunningham: "A Laboratory for Teaching O-O Thinking", OOPSLA '89 Proceedings, pages 1-6.

22.5 REUSE

The easiest and most productive way of finding classes is not to have to invent them yourself, but to get them from a library, pre-written by other designers and pre-validated by the experience of earlier reusers.

The bottom-up component

The bottom-up nature of object-oriented development should apply throughout the software development process, starting with analysis. An approach that solely focuses on the requirements document and user requests (as reflected for example by use cases) is bound to lead to a one-of-a-kind system that will be expensive to build and may miss important insights obtained by previous projects. It is part of the task of a development team, beginning at the requirements capture phase, to look at what is already available and see how existing classes may help with the new development — even if, in some cases, this means adapting the original requirements.

Too often, when we talk about finding classes, we mean *devising* them. With the development of object technology, the growth of quality libraries and the penetration of reusability ideas, *finding* will more and more retain the dictionary's sense of *coming across*.

Class wisdom

There used to live in the province of Ood a young man who longed to know the secret of finding classes. He had approached all the local masters, but none of them knew.

Having attended the public penance of Yu-Ton, a former abbot of the Sacred Order of Arrows and Bubbles, he thought that perhaps this could mean the end of his search. Upon entering Yu's cell, however, he found him still trying to understand the difference between Classes and Objects. Realizing that no enlightenment would come from there, he left without asking any questions.

On his way home he overheard two donkey-cart pushers whispering about a famous elder who was said to know the secret of classes. The next day he set out to find that great Master. Many a road he walked, many a hill he climbed, many a stream he crossed, until at last he reached the Master's hideout. By then he had searched for so long that he was no longer a young man; but like all other pilgrims he had to undergo the thirty-three-month purification rite before being permitted to meet the object of his quest.

Finally, one black winter day as the snow was savagely hitting all the surrounding mountain peaks, he was admitted into the Master's audience room. With his heart beating at the pace of a boulder rolling down the bed of a dried-up torrent, he faintly uttered his question: "Master, how can I find the classes?".

The old sage lowered her head and answered in a slow, quiet tone. "Go back to where you came from. The classes were already there."

So stunned was the questioner that it took him a few moments to notice that the Master's attendants were already whisking her away. He barely had time to run after the frail figure now disappearing forever. "Master", he asked again (almost shouting this time), "Just one more question! Please! Tell me how this story is called!"

The old Teacher tiredly turned back her head. "Should you not already know? It is the story of reuse."

22.6 THE METHOD FOR OBTAINING CLASSES

Touch by touch, the ideas discussed in this chapter amount to what we may not too pretentiously call (provided we remember that a method is a way to incubate, nurture, channel and develop invention, not a substitute for invention) the method for obtaining the classes in object-oriented software construction.

The method recognizes that class identification requires two inextricably related activities: coming up with class suggestions; and weeding out the less promising among them. The two tables which follow summarize what we have learned about these two activities. Only a few of the entries cover specific kinds of class, such as analysis classes; the rest of the advice is applicable to all cases.

First, sources of class ideas:

Sources of possible classes

Source of ideas	What to look for
Existing libraries	• Classes that address needs of the application. • Classes that describe concepts relevant to the application.
Requirements document	• Terms that occur frequently. • Terms to which the text devotes explicit definitions. • Terms that are not defined precisely but taken for granted throughout the presentation. • (Disregard grammatical categories.)
Discussions with customers and future users	• Important abstractions of the application domain. • Specific jargon of the application domain. • Remember that classes coming from the "external world" can describe *conceptual* objects as well as *material* objects.
Documentation (such as user manuals) for other systems (e.g. from competitors) in the same domain	• Important abstractions of the application domain. • Specific jargon of the application domain. • Useful design abstractions
Non-O-O systems or system descriptions	• Data elements that are passed as arguments between various components of the software, especially if they travel far. • Shared memory areas (*COMMON* blocks in Fortran). • Important files. • *DATA DIVISION* units (Cobol). • Record types (Pascal), structures and structure types (C, C++), playing an important role in the software, in particular if they are used by various routines or modules (files in C). • Entities in ER modeling.
Discussions with experienced designers	• Design classes having been successfully used in previous developments of a similar nature.
Algorithms and data structure literature	• Known data structures supporting efficient algorithms.
O-O design literature	• Applicable design patterns.

Then, criteria for investigating potential classes more carefully, and possibly rejecting them:

Reasons for rejecting a candidate class

Danger signal	Why suspicious
Class with verbal name (infinitive or imperative)	• May be a simple subroutine, not a class.
Fully effective class with only one exported routine	• May be a simple subroutine, not a class.
Class described as "performing" something	• May not be a proper data abstraction.
Class with no routine	• May be an opaque piece of information, not an ADT. Or may be an ADT, the routines having just been missed.
Class introducing no or very few features (but inherits features from parents)	• May be a case of "taxomania".
Class covering several abstractions	• Should be split into several classes, one per abstraction

22.7 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Identifying the classes is one of the principal tasks of object-oriented software construction.
- To identify the classes is a dual process: class suggestion *and* class rejection. Just as important as identifying potential class candidates is the need to eliminate unsuitable ideas.

- To identify the classes is to identify the relevant abstractions in the modeled domain and the solution space.
- "Underlining the nouns in the requirements document" is not a sufficient technique for finding the classes, since its results are too dependent on stylistic issues. It may cause designers both to miss useful classes and to include unnecessary ones.
- A broad characterization of classes distinguishes analysis classes, tied to concepts of the external world being modeled, design classes, describing architectural decisions, and implementation classes, describing data structures and algorithms.
- Design classes tend to be the most difficult to invent.
- In designing external classes, remember that external objects include concepts as well as material things.
- To decide whether a certain notion justifies defining an associated class, apply the criteria of data abstraction.
- Implementation classes include both effective classes and their deferred counterparts, describing abstract categories of implementation techniques.
- Inheritance provides a way to reuse previous designs while adapting them.
- A way to obtain classes is to evaluate candidate designs and look for any unrecognized abstraction, in particular by analyzing inter-module data transmission.
- Use cases, or scenarios, may be useful as a validation tool and as a guide to finalize an implementation, but should not be used as an analysis and design mechanism.
- The best source of classes is reusable libraries.

22.8 BIBLIOGRAPHICAL NOTES

The advice to use nouns from the requirements as a starting point for finding object types was made popular by [Booch 1986], which credits the idea to an earlier article by Abbott. Further advice appears in [Wirfs-Brock 1990].

Russell J. Abbott in Comm. ACM, 26, 11, Nov. 1983, pp. 882-894.

An article on formal specification [M 1985a] analyzes the problems raised by natural-language requirements documents. Working from a short natural-language problem description which has been used extensively in the program verification literature, it identifies a large number of deficiencies and offers a taxonomy of such deficiencies (noise, ambiguity, contradiction, remorse, overspecification, forward reference); it discusses how formal specifications can remedy some of the problems.

[Waldén 1995] presents useful advice for identifying classes.

Appendix B of [Page-Jones 1995] lists numerous "problem symptoms" in candidate object-oriented designs (for example "*class interface supports illegal or dangerous behaviors*"), alerting designers to danger signals such as have been pointed out in the present chapter. The table, as well as the rest of Page-Jones's book, offers suggestions for correcting design deficiencies.

[Ong 1993] describes a tool for converting non-O-O programs (essentially Fortran) to an object-oriented form. The conversion is semi-automatic, that is to say relies on some manual effort. Relevant to the present chapter is the authors' description of some of the heuristics they use for identifying potential classes through analysis of the original code, in particular by looking at *COMMON* blocks.

Simula 1 (the simulation language that led to modern versions of Simula) is described in [Dahl 1966]. See chapter 35 for more Simula references.

Typical data structures books, providing a precious source of implementation classes, include Knuth's famous treatise [Knuth 1968] [Knuth 1981] [Knuth 1973] and numerous college textbooks such as [Aho 1974] [Aho 1983].

A recent text, [Gore 1996], presents fundamental data structures and algorithms in a thoroughly object-oriented way.

Sources of design classes include [Gamma 1995], presenting a number of "design patterns" for C++, and [M 1994a], a compendium of library design techniques and reusable classes, discussing in detail the notions of "handle class" and "iterator class". [Krief 1996] presents the Smalltalk MVC model.

EXERCISES

E22.1 Floors as integers

See "Is a new class necessary?", page 721.

Show how to define a class *FLOOR* as heir to *INTEGER*, restricting the applicable operations.

E22.2 Inspecting objects

Daniel Halbert and Patrick O'Brien discuss the following problem, arising in the design of software development environments:

> *Consider the design of an* **inspector** *facility, used to display information about an object in a debugger window*: *the contents of its fields, and perhaps some computed values. Different kinds of inspector are needed for different object types. For instance, all the relevant information about a point can be displayed at once in a simple format, while a large two-dimensional array might best be displayed as a matrix scrollable horizontally and vertically*.
>
> *You should first decide where to put the behavior of the inspector*: *in the* [*generating class*] *of the object to be inspected or in a new, separate class*?

From [Halbert 1987], *slightly abridged*.

Answer this question by considering the pros and cons of various alternatives. (*Note*: the inheritance-related discussions of the following chapters may be useful.)