2

Criteria of object orientation

In the previous chapter we explored the goals of the object-oriented method. As a preparation for parts B and C, in which we will discover the technical details of the method, it is useful to take a quick but wide glance at the key aspects of object-oriented development. Such is the aim of this chapter.

One of the benefits will be to obtain a concise memento of what makes a system object-oriented. This expression has nowadays become so indiscriminately used that we need a list of precise properties under which we can assess any method, language or tool that its proponents claim to be O-O.

This chapter limits its explanations to a bare minimum, so if this is your first reading you cannot expect to understand in detail all the criteria listed; explaining them is the task of the rest of the book. Consider this discussion a preview — not the real movie, just a trailer.

Warning: *SPOILER*!

Actually a warning is in order because unlike any good trailer this chapter is also what film buffs call a *spoiler* — it gives away some of the plot early. As such it breaks the step-by-step progression of this book, especially part B, which patiently builds the case for object technology by looking at issue after issue before deducing and justifying the solutions. If you like the idea of reading a broad overview before getting into more depth, this chapter is for you. But if you prefer *not* to spoil the pleasure of seeing the problems unfold and of discovering the solutions one by one, then you should simply skip it. You will not need to have read it to understand subsequent chapters.

2.1 ON THE CRITERIA

Let us first examine the choice of criteria for assessing objectness.

How dogmatic do we need to be?

The list presented below includes all the facilities which I believe to be essential for the production of quality software using the object-oriented method. It is ambitious and may appear uncompromising or even dogmatic. What conclusion does this imply for an environment which satisfies some but not all of these conditions? Should one just reject such a half-hearted O-O environment as totally inadequate?

Only you, the reader, can answer this question relative to your own context. Several reasons suggest that some compromises may be necessary:

- "Object-oriented" is not a boolean condition: environment A, although not 100% O-O, may be "more" O-O than environment B; so if external constraints limit your choice to A and B you will have to pick A as the least bad object-oriented choice.
- Not everyone will need all of the properties all the time.
- Object orientation may be just one of the factors guiding your search for a software solution, so you may have to balance the criteria given here with other considerations.

All this does not change the obvious: to make informed choices, even if practical constraints impose less-than-perfect solutions, you need to know the complete picture, as provided by the list below.

Categories

The set of criteria which follows has been divided into three parts:

- *Method and language*: these two almost indistinguishable aspects cover the thought processes and the notations used to analyze and produce software. Be sure to note that (especially in object technology) the term "language" covers not just the programming language in a strict sense, but also the notations, textual or graphical, used for analysis and design.
- *Implementation and environment*: the criteria in this category describe the basic properties of the tools which allow developers to apply object-oriented ideas.
- *Libraries*: object technology relies on the reuse of software components. Criteria in this category cover both the availability of basic libraries and the mechanisms needed to use libraries and produce new ones.

This division is convenient but not absolute, as some criteria straddle two or three of the categories. For example the criterion labeled "memory management" has been classified under method and language because a language can support or prevent automatic garbage collection, but it also belongs to the implementation and environment category; the "assertion" criterion similarly includes a requirement for supporting tools.

2.2 METHOD AND LANGUAGE

The first set of criteria covers the method and the supporting notation.

Seamlessness

The object-oriented approach is ambitious: it encompasses the entire software lifecycle. When examining object-oriented solutions, you should check that the method and language, as well as the supporting tools, apply to analysis and design as well as implementation and maintenance. The language, in particular, should be a vehicle for thought which will help you through all stages of your work.

The result is a seamless development process, where the generality of the concepts and notations helps reduce the magnitude of the transitions between successive steps in the lifecycle.

These requirements exclude two cases, still frequently encountered but equally unsatisfactory:

- The use of object-oriented concepts for analysis and design only, with a method and notation that cannot be used to write executable software.
- The use of an object-oriented programming language which is not suitable for analysis and design.

In summary:

> An object-oriented language and environment, together with the supporting method, should apply to the entire lifecycle, in a way that minimizes the gaps between successive activities.

Classes

The object-oriented method is based on the notion of class. Informally, a class is a software element describing an abstract data type and its partial or total implementation. An abstract data type is a set of objects defined by the list of operations, or *features*, applicable to these objects, and the properties of these operations.

> The method and the language should have the notion of class as their central concept.

Assertions

The features of an abstract data type have formally specified properties, which should be reflected in the corresponding classes. Assertions — routine preconditions, routine postconditions and class invariants — play this role. They describe the effect of features on objects, independently of how the features have been implemented.

Assertions have three major applications: they help produce reliable software; they provide systematic documentation; and they are a central tool for testing and debugging object-oriented software.

> The language should make it possible to equip a class and its features with assertions (preconditions, postconditions and invariants), relying on tools to produce documentation out of these assertions and, optionally, monitor them at run time.

In the society of software modules, with classes serving as the cities and instructions (the actual executable code) serving as the executive branch of government, assertions provide the legislative branch. We shall see below who takes care of the judicial system.

Classes as modules

Object orientation is primarily an architectural technique: its major effect is on the modular structure of software systems.

The key role here is again played by classes. A class describes not just a type of objects but also a modular unit. In a pure object-oriented approach:

> Classes should be the only modules.

In particular, there is no notion of main program, and subprograms do not exist as independent modular units. (They may only appear as part of classes.) There is also no need for the "packages" of languages such as Ada, although we may find it convenient for management purposes to group classes into administrative units, called *clusters*.

Classes as types

The notion of class is powerful enough to avoid the need for any other typing mechanism:

> Every type should be based on a class.

Even basic types such as *INTEGER* and *REAL* can be derived from classes; normally such classes will be built-in rather than defined anew by each developer.

Feature-based computation

In object-oriented computation, there is only one basic computational mechanism: given a certain object, which (because of the previous rule) is always an instance of some class, call a feature of that class on that object. For example, to display a certain window on a screen, you call the feature *display* on an object representing the window — an instance of class *WINDOW*. Features may also have arguments: to increase the salary of an employee *e* by *n* dollars, effective at date *d*, you call the feature *raise* on *e*, with *n* and *d* as arguments.

Just as we treat basic types as predefined classes, we may view basic operations (such as addition of numbers) as special, predefined cases of feature call, a very general mechanism for describing computations:

> Feature call should be the primary computational mechanism.

A class which contains a call to a feature of a class *C* is said to be a **client** of *C*. Feature call is also known as **message passing**; in this terminology, a call such as the above will be described as passing to *e* the message "raise your pay", with arguments *d* and *n*.

Information hiding

When writing a class, you will sometimes have to include a feature which the class needs for internal purposes only: a feature that is part of the implementation of the class, but not of its interface. Others features of the class — possibly available to clients — may call the feature for their own needs; but it should not be possible for a client to call it directly.

The mechanism which makes certain features unfit for clients' calls is called information hiding. As explained in a later chapter, it is essential to the smooth evolution of software systems.

In practice, it is not enough for the information hiding mechanism to support exported features (available to all clients) and secret features (available to no client); class designers must also have the ability to export a feature selectively to a set of designated clients.

> It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.

An immediate consequence of this rule is that communication between classes should be strictly limited. In particular, a good object-oriented language should not offer any notion of global variable; classes will exchange information exclusively through feature calls, and through the inheritance mechanism.

Exception handling

Abnormal events may occur during the execution of a software system. In object-oriented computation, they often correspond to calls that cannot be executed properly, as a result of a hardware malfunction, of an unexpected impossibility (such as numerical overflow in an addition), or of a bug in the software.

To produce reliable software, it is necessary to have the ability to recover from such situations. This is the purpose of an exception mechanism.

> The language should provide a mechanism to recover from unexpected abnormal situations.

In the society of software systems, as you may have guessed, the exception mechanism is the third branch of government, the judicial system (and the supporting police force).

Static typing

When the execution of a software system causes the call of a certain feature on a certain object, how do we know that this object will be able to handle the call? (In message terminology: how do we know that the object can process the message?)

To provide such a guarantee of correct execution, the language must be typed. This means that it enforces a few compatibility rules; in particular:

- Every entity (that is to say, every name used in the software text to refer to run-time objects) is explicitly declared as being of a certain type, derived from a class.
- Every feature call on a certain entity uses a feature from the corresponding class (and the feature is available, in the sense of information hiding, to the caller's class).
- Assignment and argument passing are subject to **conformance rules**, based on inheritance, which require the source's type to be compatible with the target's type.

In a language that imposes such a policy, it is possible to write a **static type checker** which will accept or reject software systems, guaranteeing that the systems it accepts will not cause any "feature not available on object" error at run time.

> A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.

Genericity

For typing to be practical, it must be possible to define type-parameterized classes, known as generic. A generic class *LIST* [*G*] will describe lists of elements of an arbitrary type represented by *G*, the "formal generic parameter"; you may then declare specific lists through such derivations as *LIST* [*INTEGER*] and *LIST* [*WINDOW*], using types *INTEGER* and *WINDOW* as "actual generic parameters". All derivations share the same class text.

> It should be possible to write classes with formal generic parameters representing arbitrary types.

This form of type parameterization is called **unconstrained** genericity. A companion facility mentioned below, constrained genericity, involves inheritance.

Single inheritance

Software development involves a large number of classes; many are variants of others. To control the resulting potential complexity, we need a classification mechanism, known as inheritance. A class will be an heir of another if it incorporates the other's features in addition to its own. (A *descendant* is a direct or indirect heir; the reverse notion is *ancestor*.)

> It should be possible to define a class as inheriting from another.

Inheritance is one of the central concepts of the object-oriented methods and has profound consequences on the software development process.

Multiple inheritance

We will often encounter the need to combine several abstractions. For example a class might model the notion of "infant", which we may view both as a "person", with the

associated features, and, more prosaically, as a "tax-deductible item", which earns some deduction at tax time. Inheritance is justified in both cases. *Multiple* inheritance is the guarantee that a class may inherit not just from one other but from as many as is conceptually justified.

Multiple inheritance raises a few technical problems, in particular the resolution of *name clashes* (cases in which different features, inherited from different classes, have the same name). Any notation offering multiple inheritance must provide an adequate solution to these problems.

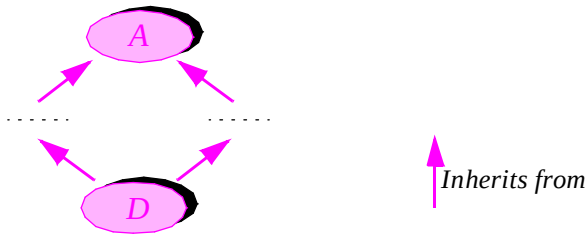
> It should be possible for a class to inherit from as many others as necessary, with an adequate mechanism for disambiguating name clashes.

The solution developed in this book is based on *renaming* the conflicting features in the heir class.

Repeated inheritance

Multiple inheritance raises the possibility of *repeated* inheritance, the case in which a class inherits from another through two or more paths, as shown.

Repeated inheritance



In such a case the language must provide precise rules defining what happens to features inherited repeatedly from the common ancestor, *A* in the figure. As the discussion of repeated inheritance will show, it may be desirable for a feature of *A* to yield just one feature of *D* in some cases (*sharing*), but in others it should yield two (*replication*). Developers must have the flexibility to prescribe either policy separately for each feature.

> Precise rules should govern the fate of features under repeated inheritance, allowing developers to choose, separately for each repeatedly inherited feature, between sharing and replication.

Constrained genericity

The combination of genericity and inheritance brings about an important technique, constrained genericity, through which you can specify a class with a generic parameter that represents not an arbitrary type as with the earlier (unconstrained) form of genericity, but a type that is a descendant of a given class.

A generic class *SORTABLE_LIST*, describing lists with a *sort* feature that will reorder them sequentially according to a certain order relation, needs a generic parameter representing the list elements' type. That type is not arbitrary: it must support an order relation. To state that any actual generic parameter must be a descendant of the library class *COMPARABLE*, describing objects equipped with an order relation, use constrained genericity to declare the class as *SORTABLE_LIST* [*G –> COMPARABLE*].

> The genericity mechanism should support the constrained form of genericity.

Redefinition

When a class is an heir of another, it may need to change the implementation or other properties of some of the inherited features. A class *SESSION* describing user sessions in an operating system may have a feature *terminate* to take care of cleanup operations at the end of a session; an heir might be *REMOTE_SESSION*, handling sessions started from a different computer on a network. If the termination of a remote session requires supplementary actions (such as notifying the remote computer), class *REMOTE_SESSION* will redefine feature *terminate*.

Redefinition may affect the implementation of a feature, its signature (type of arguments and result), and its specification.

> It should be possible to redefine the specification, signature and implementation of an inherited feature.

Polymorphism

With inheritance brought into the picture, the static typing requirement listed earlier would be too restrictive if it were taken to mean that every entity declared of type *C* may only refer to objects whose type is exactly *C*. This would mean for example that an entity of type *C* (in a navigation control system) could not be used to refer to an object of type *MERCHANT_SHIP* or *SPORTS_BOAT*, both assumed to be classes inheriting from *BOAT*.

> As noted earlier, an "entity" is a name to which various values may become attached at run time. This is a generalization of the traditional notion of variable.

Polymorphism is the ability for an entity to become attached to objects of various possible types. In a statically typed environment, polymorphism will not be arbitrary, but controlled by inheritance; for example, we should not allow our *BOAT* entity to become

attached to an object representing an object of type *BUOY*, a class which does not inherit from *BOAT*.

> It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.

Dynamic binding

The combination of the last two mechanisms mentioned, redefinition and polymorphism, immediately suggests the next one. Assume a call whose target is a polymorphic entity, for example a call to the feature *turn* on an entity declared of type *BOAT*. The various descendants of *BOAT* may have redefined the feature in various ways. Clearly, there must be an automatic mechanism to guarantee that the version of *turn* will always be the one deduced from the actual object's type, regardless of how the entity has been declared. This property is called dynamic binding.

> Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.

Dynamic binding has a major influence on the structure of object-oriented applications, as it enables developers to write simple calls (meaning, for example, "call feature *turn* on entity *my_boat*") to denote what is actually several possible calls depending on the corresponding run-time situations. This avoids the need for many of the repeated tests ("Is this a merchant ship? Is this a sports boat?") which plague software written with more conventional approaches.

Run-time type interrogation

Object-oriented software developers soon develop a healthy hatred for any style of computation based on explicit choices between various types for an object. Polymorphism and dynamic binding provide a much preferable alternative. In some cases, however, an object comes from the outside, so that the software author has no way to predict its type with certainty. This occurs in particular if the object is retrieved from external storage, received from a network transmission or passed by some other system.

The software then needs a mechanism to access the object in a safe way, without violating the constraints of static typing. Such a mechanism should be designed with care, so as not to cancel the benefits of polymorphism and dynamic binding.

The **assignment attempt** operation described in this book satisfies these requirements. An assignment attempt is a conditional operation: it tries to attach an object to an entity; if in a given execution the object's type conforms to the type declared for the entity, the effect is that of a normal assignment; otherwise the entity gets a special "void"

value. So you can handle objects whose type you do not know for sure, without violating the safety of the type system.

> It should be possible to determine at run time whether the type of an object conforms to a statically given type.

Deferred features and classes

In some cases for which dynamic binding provides an elegant solution, obviating the need for explicit tests, there is no initial version of a feature to be redefined. For example class *BOAT* may be too general to provide a default implementation of *turn*. Yet we want to be able to call feature *turn* to an entity declared of type *BOAT* if we have ensured that at run time it will actually be attached to objects of such fully defined types as *MERCHANT_SHIP* and *SPORTS_BOAT*.

In such cases *BOAT* may be declared as a deferred class (one which is not fully implemented), and with a deferred feature *turn*. Deferred features and classes may still possess assertions describing their abstract properties, but their implementation is postponed to descendant classes. A non-deferred class is said to be *effective*.

> It should be possible to write a class or a feature as deferred, that is to say specified but not fully implemented.

Deferred classes (also called abstract classes) are particularly important for object-oriented analysis and high-level design, as they make it possible to capture the essential aspects of a system while leaving details to a later stage.

Memory management and garbage collection

The last point on our list of method and language criteria may at first appear to belong more properly to the next category — implementation and environment. In fact it belongs to both. But the crucial requirements apply to the language; the rest is a matter of good engineering.

Object-oriented systems, even more than traditional programs (except in the Lisp world), tend to create many objects with sometimes complex interdependencies. A policy leaving developers in charge of managing the associated memory, especially when it comes to reclaiming the space occupied by objects that are no longer needed, would harm both the efficiency of the development process, as it would complicate the software and occupy a considerable part of the developers' time, and the safety of the resulting systems, as it raises the risk of improper recycling of memory areas. In a good object-oriented environment memory management will be automatic, under the control of the *garbage collector*, a component of the runtime system.

The reason this is a language issue as much as an implementation requirement is that a language that has not been explicitly designed for automatic memory management will often render it impossible. This is the case with languages where a pointer to an object of

a certain type may disguise itself (through conversions known as "casts") as a pointer of another type or even as an integer, making it impossible to write a safe garbage collector.

> The language should make safe automatic memory management possible, and the implementation should provide an automatic memory manager taking care of garbage collection.

2.3 IMPLEMENTATION AND ENVIRONMENT

We come now to the essential features of a development environment supporting object-oriented software construction.

Automatic update

Software development is an incremental process. Developers do not commonly write thousands of lines at a time; they proceed by addition and modification, starting most of the time from a system that is already of substantial size.

When performing such an update, it is essential to have the guarantee that the resulting system will be consistent. For example, if you change a feature *f* of class *C*, you must be certain that every descendant of *C* which does not redefine *f* will be updated to have the new version of *f*, and that every call to *f* in a client of *C* or of a descendant of *C* will trigger the new version.

Conventional approaches to this problem are manual, forcing the developers to record all dependencies, and track their changes, using special mechanisms known as "make files" and "include files". This is unacceptable in modern software development, especially in the object-oriented world where the dependencies between classes, resulting from the client and inheritance relations, are often complex but may be deduced from a systematic examination of the software text.

> System updating after a change should be automatic, the analysis of inter-class dependencies being performed by tools, not manually by developers.

It is possible to meet this requirement in a compiled environment (where the compiler will work together with a tool for dependency analysis), in an interpreted environment, or in one combining both of these language implementation techniques.

Fast update

In practice, the mechanism for updating the system after some changes should not only be automatic, it should also be fast. More precisely, it should be proportional to the size of

the changed parts, not to the size of the system as a whole. Without this property, the method and environment may be applicable to small systems, but not to large ones.

> The time to process a set of changes to a system, enabling execution of the updated version, should be a function of the size of the changed components, independent of the size of the system as a whole.

Here too both interpreted and compiled environments may meet the criterion, although in the latter case the compiler must be incremental. Along with an incremental compiler, the environment may of course include a global optimizing compiler working on an entire system, as long as that compiler only needs to be used for delivering a final product; development will rely on the incremental compiler.

Persistence

Many applications, perhaps most, will need to conserve objects from one session to the next. The environment should provide a mechanism to do this in a simple way.

An object will often contain references to other objects; since the same may be true of these objects, this means that every object may have a large number of *dependent* objects, with a possibly complex dependency graph (which may involve cycles). It would usually make no sense to store or retrieve the object without all its direct and indirect dependents. A persistence mechanism which can automatically store an object's dependents along with the object is said to support **persistence closure**.

> A persistent storage mechanism supporting persistence closure should be available to store an object and all its dependents into external devices, and to retrieve them in the same or another session.

For some applications, mere persistence support is not sufficient; such applications will need full **database support**. The notion of object-oriented database is covered in a later chapter, which also explores other persistent issues such as *schema evolution*, the ability to retrieve objects safely even if the corresponding classes have changed.

Documentation

Developers of classes and systems must provide management, customers and other developers with clear, high-level descriptions of the software they produce. They need tools to assist them in this effort; as much as possible of the documentation should be produced automatically from the software texts. Assertions, as already noted, help make such software-extracted documents precise and informative.

> Automatic tools should be available to produce documentation about classes and systems.

Browsing

When looking at a class, you will often need to obtain information about other classes; in particular, the features used in a class may have been introduced not in the class itself but in its various ancestors. This puts on the environment the burden of providing developers with tools to examine a class text, find its dependencies on other classes, and switch rapidly from one class text to another.

S is a "supplier" of C if C is a client of S. "Client" was defined on page 24.

This task is called browsing. Typical facilities offered by good browsing tools include: find the clients, suppliers, descendants, ancestors of a class; find all the redefinitions of a feature; find the original declaration of a redefined feature.

> Interactive browsing facilities should enable software developers to follow up quickly and conveniently the dependencies between classes and features.

2.4 LIBRARIES

One of the characteristic aspects of developing software the object-oriented way is the ability to rely on libraries. An object-oriented environment should provide good libraries, and mechanisms to write more.

Basic libraries

The fundamental data structures of computing science — sets, lists, trees, stacks… — and the associated algorithms — sorting, searching, traversing, pattern matching — are ubiquitous in software development. In conventional approaches, each developer implements and re-implements them independently all the time; this is not only wasteful of efforts but detrimental to software quality, as it is unlikely that an individual developer who implements a data structure not as a goal in itself but merely as a component of some application will attain the optimum in reliability and efficiency.

An object-oriented development environment must provide reusable classes addressing these common needs of software systems.

> Reusable classes should be available to cover the most frequently needed data structures and algorithms.

Graphics and user interfaces

Many modern software systems are interactive, interacting with their users through graphics and other pleasant interface techniques. This is one of the areas where the object-oriented model has proved most impressive and helpful. Developers should be able to rely on graphical libraries to build interactive applications quickly and effectively.

> Reusable classes should be available for developing applications which provide their users with pleasant graphical user interface.

Library evolution mechanisms

Developing high-quality libraries is a long and arduous task. It is impossible to guarantee that the design of library will be perfect the first time around. An important problem, then, is to enable library developers to update and modify their designs without wreaking havoc in existing systems that depend on the library. This important criterion belongs to the library category, but also to the method and language category.

> Mechanisms should be available to facilitate library evolution with minimal disruption of client software.

Library indexing mechanisms

Another problem raised by libraries is the need for mechanisms to identify the classes addressing a certain need. This criterion affects all three categories: libraries, language (as there must be a way to enter indexing information within the text of each class) and tools (to process queries for classes satisfying certain conditions).

> Library classes should be equipped with indexing information allowing property-based retrieval.

2.5 FOR MORE SNEAK PREVIEW

Although to understand the concepts in depth it is preferable to read this book sequentially, readers who would like to complement the preceding theoretical overview with an advance glimpse of the method at work on a practical example can at this point read chapter 20, a case study of a practical design problem, on which it compares an O-O solution with one employing more traditional techniques.

That case study is mostly self-contained, so that you will understand the essentials without having read the intermediate chapters. (But if you do go ahead for this quick peek, you must promise to come back to the rest of the sequential presentation, starting with chapter 3, as soon as you are done.)

2.6 BIBLIOGRAPHICAL NOTES AND OBJECT RESOURCES

This introduction to the criteria of object orientation is a good opportunity to list a selection of books that offer quality introductions to object technology in general.

[Waldén 1995] discusses the most important issues of object technology, focusing on analysis and design, on which it is probably the best reference.

[Page-Jones 1995] provides an excellent overview of the method.

[Cox 1990] (whose first edition was published in 1986) is based on a somewhat different view of object technology and was instrumental in bringing O-O concepts to a much larger audience than before.

[Henderson-Sellers 1991] (a second edition is announced) provides a short overview of O-O ideas. Meant for people who are asked by their company to "go out and find out what that object stuff is about", it includes ready-to-be-photocopied transparency masters, precious on such occasions. Another overview is [Eliëns 1995].

The *Dictionary of Object Technology* [Firesmith 1995] provides a comprehensive reference on many aspects of the method.

All these books are to various degrees intended for technically-minded people. There is also a need to educate managers. [M 1995] grew out of a chapter originally planned for the present book, which became a full-fledged discussion of object technology for executives. It starts with a short technical presentation couched in business terms and continues with an analysis of management issues (lifecycle, project management, reuse policies). Another management-oriented book, [Goldberg 1995], provides a complementary perspective on many important topics. [Baudoin 1996] stresses lifecycle issues and the importance of standards.

Coming back to technical presentations, three influential books on object-oriented languages, written by the designers of these languages, contain general methodological discussions that make them of interest to readers who do not use the languages or might even be critical of them. (The history of programming languages and books about them shows that designers are not always the best to write about their own creations, but in these cases they were.) The books are:

- *Simula BEGIN* [Birtwistle 1973]. (Here two other authors joined the language designers Nygaard and Dahl.)
- *Smalltalk-80*: *The Language and its Implementation* [Goldberg 1983].
- *The C++ Programming Language, second edition* [Stroustrup 1991].

Chapter 29 discusses teaching the technology.

More recently, some introductory programming textbooks have started to use object-oriented ideas right from the start, as there is no reason to let "ontogeny repeat phylogeny", that is to say, take the poor students through the history of the hesitations and mistakes through which their predecessors arrived at the right ideas. The first such text (to my knowledge) was [Rist 1995]. Another good book covering similar needs is [Wiener 1996]. At the next level — textbooks for a second course on programming, discussing data structures and algorithms based on the notation of this book — you will find [Gore 1996] and [Wiener 1997]; [Jézéquel 1996] presents the principles of object-oriented software engineering.

The Usenet newsgroup *comp.object*, archived on several sites around the Web, is the natural medium of discussion for many issues of object technology. As with all such forums, be prepared for a mixture of the good, the bad and the ugly. The Object Technology department of *Computer* (IEEE), which I have edited since it started in 1995, has frequent invited columns by leading experts.

Magazines devoted to Object Technology include:

- The *Journal of Object-Oriented Programming* (the first journal in the field, emphasizing technical discussions but for a large audience), *Object Magazine* (of a more general scope, with some articles for managers), *Objekt Spektrum* (German), *Object Currents* (on-line), all described at *http://www.sigs.com*.
- *Theory and Practice of Object Systems*, an archival journal.
- *L'OBJET* (French), described at *http://www.tools.com/lobjet*.

The major international O-O conferences are OOPSLA (yearly, USA or Canada, see *http://www.acm.org*); *Object Expo* (variable frequency and locations, described at *http://www.sigs.com*); and TOOLS (Technology of Object-Oriented Languages and Systems), organized by ISE with three sessions a year (USA, Europe, Pacific), whose home page at *http://www.tools.com* also serves as a general resource on object technology and the topics of this book.