28

The software construction process

Foremost among the methodological issues of object technology is how it affects the broader picture of software development. We will now examine the consequences of object-oriented principles on the organization of projects and their division into phases.

[M 1995].

Such a presentation is part of a more general topic: the management perspective on object technology. Another book, *Object Success*, explores management issues in detail. The discussion which follows, drawing in part from *Object Success*, presents the essential ideas: **clusters**, the basic organizational unit; principles of **concurrent engineering** leading to the cluster model of the software lifecycle; steps and tasks of that model; the role of **generalization** for reusability; and the principles of **seamlessness** and **reversibility**.

28.1 CLUSTERS

The module structure of the object-oriented method is the class. For organizational purposes, you will usually need to group classes into collections, called clusters — a notion briefly previewed in the last chapter's sketch of the Business Object Notation.

A cluster is a group of related classes or, recursively, of related clusters.

The two cases are exclusive: for simplicity and ease of management, a cluster that contains subclusters should not have any classes of its own. So a cluster will be either a *basic cluster*, made of classes, or a *supercluster*, made of other clusters.

Typical basic clusters could include a parsing cluster for analyzing users' text input, a graphic cluster for graphical manipulations, a communications cluster. A basic cluster will typically have somewhere between five and forty classes; at around twenty classes, you should start thinking about splitting it into subclusters. The cluster is also the natural unit for single-developer mastery: each cluster should be managed by one person, and one person should be able to understand *all* of it — whereas in a large development no one can understand all of a system or even a major subsystem.

On super-modules see "The architectural role of selective exports", page 209.

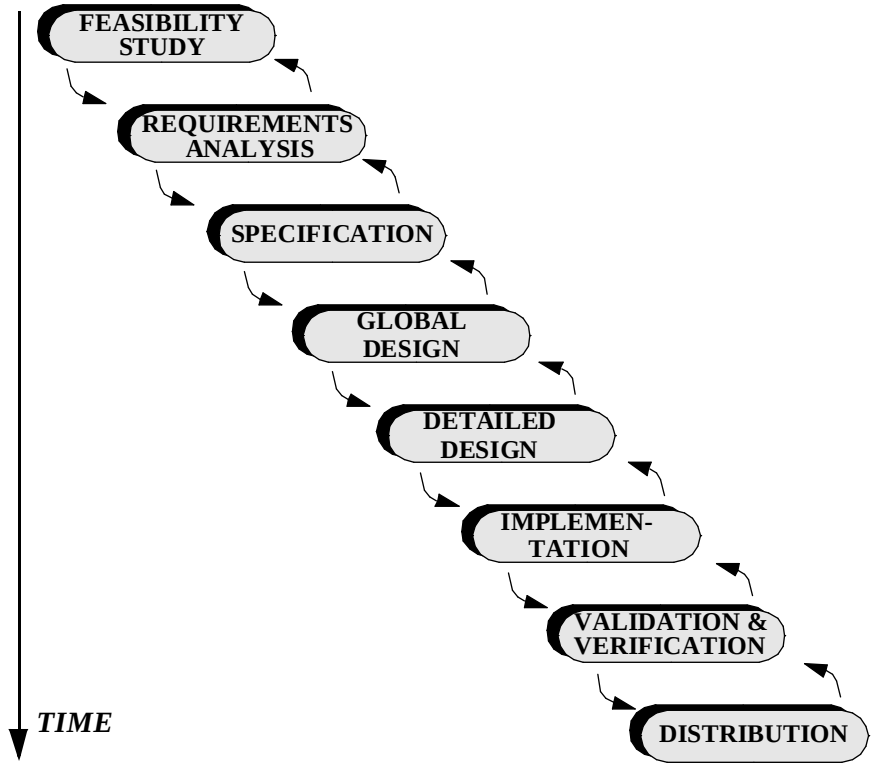
Clusters are not super-modules. In an earlier chapter we saw the arguments for avoiding the introduction of units such as packages, and instead keeping a single module mechanism, the class.

Unlike packages, clusters are not a language construct, although they will appear in the Lace control files used to assemble systems out of components. They are a management tool. The responsibility for finding clusters will rest with the project leader; less challenging than the task of finding classes, studied in detail in a previous chapter, clustering classes mostly relies on common sense and the project leader's experience. This point actually deserves some emphasis, as it is sometimes misunderstood: the truly difficult job, which can launch a project on to an auspicious life or wreck it, and for which one can talk of right and wrong solutions, is to identify the classes (the proper data abstractions); grouping these classes into clusters is an organizational matter, for which many solutions are possible, depending on the resources available and on the expertise of the various team members. A less-than-optimal clustering decision may cause trouble and slow the development, but will not by itself bring the project down.

On Lace see "Assembling a system", page 198.

28.2 CONCURRENT ENGINEERING

One of the consequences of the division into clusters is that we can avoid the disadvantages of the all-or-nothing nature of traditional software lifecycle models. The well-known "waterfall" approach, introduced in 1970, was a reaction against the "code it now and fix it later" approach of that bygone era. It had the merit of separating concerns, of defining the principal tasks of software engineering, and of emphasizing the importance of up-front specification and design tasks.



The waterfall model

(*WARNING*: *this is not the recommended process model for O-O development*!)

But the Waterfall Model also suffers (among other deficiencies) from the rigidity of its approach: taken literally, it would mean that no design can proceed until all the specification is complete, no implementation until all design is complete. This is a certain recipe for disaster: one grain of sand in the machine, and the whole project comes to a halt.

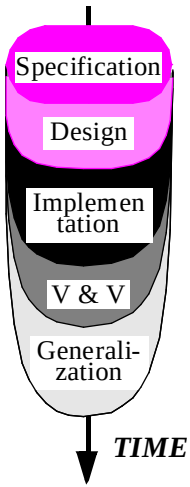
Various proposals such as the Spiral model have attempted to reduce this risk by providing a more iterative approach, But they retain the one-thread approach of the Waterfall, which hardly reflects the nature of today's software development, especially for large "virtual" teams that may be distributed over many sites, communicating through the Internet and other "electronic collocation" mechanisms.

Successful object-oriented development needs to support a **concurrent engineering** scheme, offering decentralization and flexibility, without losing the benefits of the waterfall's orderliness. We will in particular have to retain a sequential component, with well-defined activities. Object-oriented development does not mean that we can or should get rid of sound engineering practices. If anything, the added power of the method requires us to be *more* organized than before.

With a division into clusters we can achieve the right balance between sequentiality and concurrent engineering. We will have a sequential process, but subject to backward adjustments (this is the concept of reversibility, discussed in more detail at the end of this chapter), and applied to **clusters** rather than to the entire system.

The mini-lifecycle governing the development of a cluster may pictured as this:

Individual cluster lifecycle



The shape of the activity representations suggests the seamless nature of the development. Instead of separate steps as in the waterfall model, we see an accretion process — think of the figure as depicting a stalactite — in which every step takes over from the previous one and adds its own contribution.

28.3 STEPS AND TASKS

The steps listed in the mini-lifecycle of each cluster are:

- Specification: identify the classes (data abstractions) of the cluster and their major features and constraints (yielding invariant clauses).
- Design: define the architecture of the classes and their relations.
- Implementation: finalize the classes, with all details added.
- Verification & Validation: check that the cluster's classes perform satisfactorily (through static examination, testing and other techniques).
- Generalization: prepare for reuse (see below).

Given the high-level of abstraction of the method, the distinction between design and implementation is not always clear-cut. So a variant of the model merges these two steps into one, "design-implementation".

The need remains for two system-wide, cluster-independent phases. First, as with any other approach, you should perform a **feasibility study**, resulting in a go or no-go decision. Then, the project needs to be divided into clusters; this is, as noted, the responsibility of the project leader, who can of course rely on the help of other experienced team members.

28.4 THE CLUSTER MODEL OF THE SOFTWARE LIFECYCLE

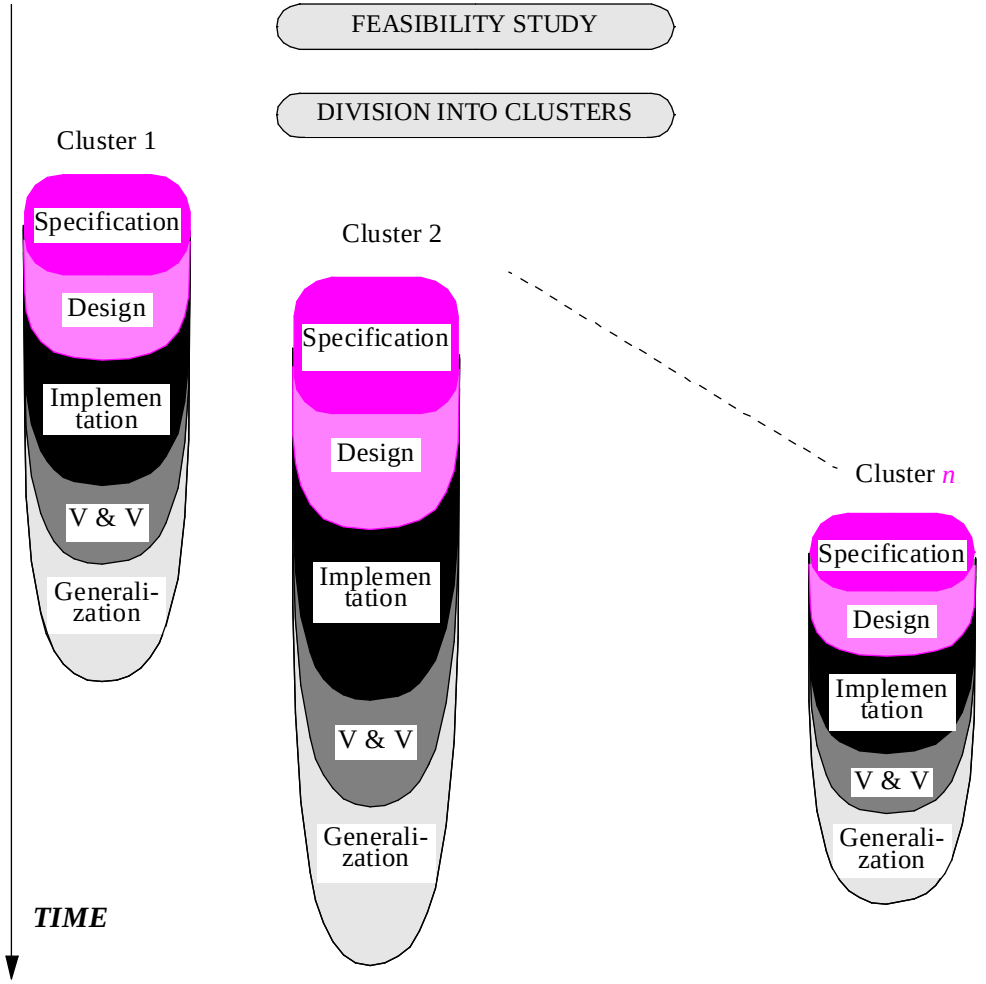
The general development scheme, known as the Cluster Model, appears on the facing page. The vertical axis represents the sequential component of the process: a step that appears lower than another will be executed after it. The horizontal direction reflects concurrent engineering: tasks at the same level can proceed in parallel.

Various clusters, and various steps within each cluster, will proceed at their own pace depending on the difficulty of the task. The project leader is in charge of deciding when to start a new cluster or a new task.

The result is to give the project leader the right combination of order and flexibility. Order because the definition of cluster tasks provides a control framework and control points against which to assess progress and delays (one of the most difficult aspects of project management); flexibility because you can buffer unexpected delays, or take advantage of unexpectedly fast progress, by starting activities sooner or later. The project leader also controls the degree of concurrent engineering: for a small team, or in the early stages of a difficult project, there may be a small number of parallel clusters, or just one; for a larger team, or once the basic existential questions seems to be under control, you can start pursuing several clusters at once.

Better than traditional approaches, the cluster model enables project leaders to do their job to its full extent, exerting their decision power to devote resources where they are needed the most.

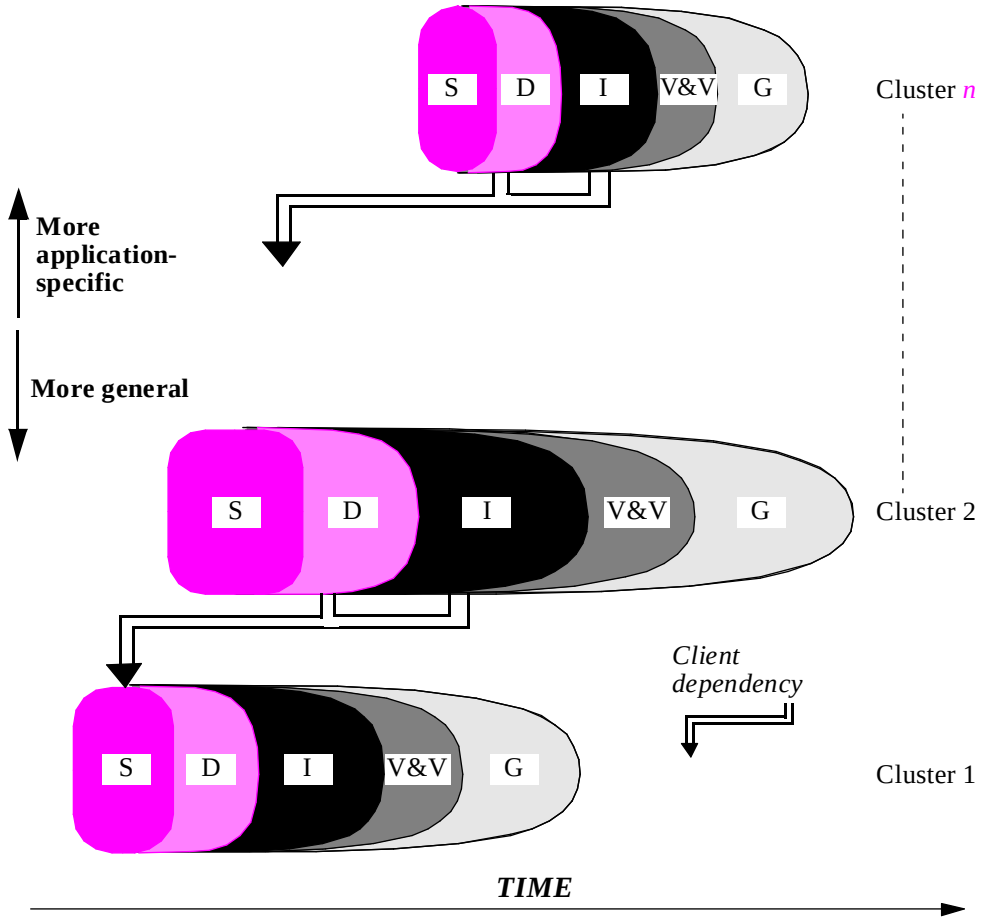
The cluster model of the software lifecycle



To avoid divergence, the current states of the various clusters' development must be regularly reconciled. This is the task of **integration**, best performed at preset intervals, for example once a week. It is the responsibility of the project leader, and ensures that at every stage after start-up there will be a **current demo**, not necessarily up to date for all aspects of the system, but ready to be showed to whoever — customers, managers... — needs reassurance about the project's progress. This also serves to remove any inconsistency between clusters before it has had the opportunity to cause damage, reassuring the project members themselves that the pieces fit together and that the future system is taking shape.

What makes possible the cluster model's form of concurrent engineering is the set of information hiding properties of the object-oriented method. Clusters may depend on each other; for example a graphical interface cluster may need, for remote display, classes of the communication cluster. Thanks to data abstraction, it is possible for a cluster to proceed even if the clusters on which it depends are not yet finished; it suffices that the *specification* phase of the needed classes be complete, so that you can proceed on the basis

of their official interface, given as a short form or deferred version. This aspect of the model is perhaps easier to picture if we rotate the preceding figure, as illustrated below, to emphasize the software layers corresponding to the various clusters, with the more general clusters at the bottom and the more application-specific ones at the top. The design and implementation of each cluster depend only on the specifications of clusters below it, not on their own design and implementation. The figure only shows dependencies on the cluster immediately below, but a cluster may rely on any lower-level cluster.



A project's clusters as a set of abstraction layers

28.5 GENERALIZATION

The last task of cluster mini-lifecycles, generalization (the G on the above figure) has no equivalent in traditional approaches. Its goal is to polish the classes so as to turn them into potentially reusable software components.

Including a generalization step immediately suggests a criticism: instead of an a posteriori add-on, should reusability concerns not be part of the entire software process? How can one make software reusable after the fact? But this criticism is misplaced. The a

priori view of software reuse ("to be reusable, software should be designed as reusable from the start") and the a posteriori view ("software will not be reusable the first time around") are complementary, not contradictory. The success of a reusability policy requires both instilling a *reusability culture* in the minds of everyone involved, and devoting sufficient resources to improving the reusability of classes' initial versions.

In spite of the best of intentions, software elements produced as part of an application-oriented project will usually not be fully reusable. This is due in part to the constraints affecting projects — the pressure of customers wanting the next version ASAP, of the competition putting out its own products, of shareholders eager to see results. We live in a hurried world and an even more hurried industry. But there is also an intrinsic reason for not always trusting reusability promises: until someone has reused it, you cannot be sure that a product has been freed of all its dependencies, explicit and (particularly) implicit, on its original developers' background, corporate affiliation, technical context, working practices, hardware resources and software environment.

The presence of a generalization step is not, then, an excuse for ignoring reusability until the last moment. The arguments of the a priori school are correct: you cannot add reusability as an afterthought. But do not assume that having a reusability policy is sufficient. Even with reusability built into everyone's mindset, you will need to devote some more time to your project's classes before you can call them software components.

Including a generalization step in the official process model is also a matter of policy. Very few corporate executives these days will take a public stand *against* reusability. Of course, my friend, we want our software to be reusable! The software people need to find out whether this is sincere commitment or lip service. Very easy. The commitment exists if management is ready to reserve *some* resources, on top of the money and time allocated to each project, for generalization. This is a courageous decision, because the benefits may not be immediate and other urgent projects may suffer a little. But it is the only way to guarantee that there will, in the end, be reusable components. If, however, the management is not ready to pledge such resources, even modest ones (a few percent above the normal project budget can make a world of difference), then you can listen politely to the grandiose speeches about reuse and read sympathetically about the "reuser of the month" award in the company's newsletter: in truth, the company is not ready for reusability and will not get reusability.

If, on the other hand, some resources are devoted to generalization, remember that this is not sufficient either. Success in reusability comes from a combination of a priori and a posteriori efforts:

The reusability culture

Develop all software under the assumption that it will be reused.

Do not trust that any software will be reusable until you have seen it reused.

The first part implies applying reusability concerns throughout development. The second implies not taking the result for granted, but performing a generalization step to remove any traces of context-specific elements.

The generalization task may involve the following activities:

- **Abstracting**: introducing a deferred class to describe the pure abstraction behind a certain class.
- **Factoring**: recognizing that two classes, originally unrelated, are in fact variants of the same general notion, which can then be described by a common ancestor.
- Adding assertions, especially postconditions and invariant clauses which reflect increased understanding of the semantics of the class and its features. (You may also have to add a precondition, but this is more akin to correcting a bug, since it means the routine was not properly protected.)
- Adding **rescue** clauses to handle exceptions whose possibility may initially have been ignored.
- Adding documentation.

On abstracting and factoring see "Varieties of class abstraction", page 860.

The first two of these activities, studied in the discussion of inheritance methodology, reflect the non-standard view of inheritance hierarchy construction that we explored then: the recognition that, although it would be nice always to go from the general to the specific and the abstract to the concrete, the actual path to invention is often more tortuous, and sometimes just the other way around.

The role of generalization is to improve classes that may be considered good enough for internal purposes — as long, that is, as they are only used within a particular system — but not any more when they become part of a library available to any client author who cares to use them for his own needs. Peccadillos that may have been forgivable in the first setting, such as insufficient specification or reliance on undocumented assumptions, become show-stoppers. This is why developing for reusability is more difficult than ordinary application development: when your software is available to anyone, working on applications of any kind for any platform anywhere in the world, everything starts to matter. Reusability breeds perfectionism; you cannot leave good enough alone.

28.6 SEAMLESSNESS AND REVERSIBILITY

The "stalactite" nature of the cluster lifecycle reflects one of the most radical differences between O-O development and earlier approaches. Instead of erecting barriers between successive lifecycle steps, well-understood object technology defines a single framework for analysis, design, implementation and maintenance. This is known as *seamless development*; one of its consequences, previewed in the last chapter's discussion of the Business Object Notation, is the need for a *reversible* software development process.

Seamless development

Different tasks will of course remain. To take extreme examples, you are not doing the same thing when defining general properties of a system that has yet to be built and performing the last rounds of debugging. But the idea of seamlessness is to downplay differences where the traditional approach exaggerated them; to recognize, behind the technical variations, the fundamental unity of the software process. Throughout development the same issues arise, the same intellectual challenges must be addressed, the same structuring mechanisms are needed, the same forms of reasoning apply and, as shown in this book, the same notation can be used.

The benefits of a seamless approach are numerous:

- You avoid costly and error-prone transitions between steps, magnified by changes in notation, mindset, and personnel (analysts, designers, implementers...). Such gaps are often called **impedance mismatches** by analogy with a circuit made of electrically incompatible elements; the mismatches between analysis and design, design and implementation, implementation and evolution, are among the worst causes of trouble in traditional software development.

"Direct Mapping", page 47.

- By starting from the analysis classes as a basis for the rest of the development, you ensure a close correspondence between the description of the problem and the solution. This **direct mapping property** helps the dialog with customers and users, and facilitates evolution by ensuring that they all think in terms of the same basic concepts. It is part of the O-O method's support for extendibility.

- The use of a single framework facilitates the backward adjustments that will inevitably accompany the normally one-directional progress of the software development process.

Reversibility: wisdom sometimes blooms late in the season

The last benefit cited defines one of the principal contributions of object technology to the software lifecycle — reversibility.

Reversibility is the official acceptance of a characteristic of software development which, although inevitable and universal, is one of the most closely guarded secrets of the software literature: the influence of later stages of the software process on decisions made during initial stages.

We all wish, of course, that problems be fully defined before we get to solve them. That is the normal way to go, and in software it means that we complete the analysis before we engage in design, the design before we start implementation, the implementation before we deliver. But what if, during implementation, a developer suddenly realizes that the system could do something better, or should do something different altogether? Do we scold him for not minding his own business? What if his suggestion is indeed right?

The phrase *esprit de l'escalier*, "wit of the staircase", captures this phenomenon. Picture a pleasant dinner in an apartment on the second or fourth floor (the fashionable ones) of a Parisian building. Sharp comments fly back and forth over the veal Marengo, and you feel dumb. The soirée finishes and you take leave of your hosts, start walking down the stairs, when … there it is: the smashing repartee that would have made you the hero of the evening! But too late.

Are bouts of *esprit de l'escalier* too late in software also? They have existed ever since software projects have been told to freeze the specification before they start on a solution. Bad managers suppress them, telling the implementers, in effect, to code and shut up. Good managers try to see whether they can take advantage of belated specification ideas, without attracting the attention of whoever is in charge of enforcing the company's software quality plan and its waterfall-style ukases against changing the specification at implementation time.

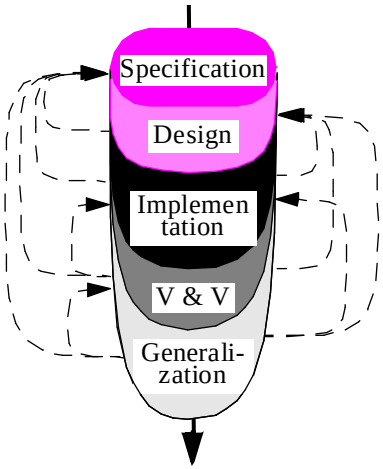
The bad managers may be unconsciously applying another **escalier** *aphorism, Clemenceau's* "in love, the best moment is in the stairs" — *beforehand, that is.*

With O-O development it becomes clear that the *esprit de l'escalier* phenomenon is not just the result of laziness in analysis, but follows from the intrinsic nature of software development. Wisdom sometimes blooms late in the season. Nowhere more than with object technology do we see the intimate connection between problem and solution that characterizes our field. It is not just that we sometimes understand aspects of the problem only at the time of the solution, but more profoundly that the solution affects the problem and suggests better functionalities.

Remember the example of command undoing and redoing: an implementation technique, the "history list" — which someone trained in a more traditional approach would dismiss as irrelevant to the task of defining system functionality —, actually suggested a new way of providing end-users of our system with a convenient interface for undoing and redoing commands.

Chapter 21.

The introduction of reversibility suggests that the general forward thrust of our earlier cluster mini-lifecycle diagrams is actually tempered by the constant possibility of backward revisions and corrections:



Individual cluster lifecycle, reversible

28.7 WITH US, EVERYTHING IS THE FACE

The stress on seamlessness and reversibility is perhaps the most potentially subversive component of object technology. It affects project organization, and the very nature of the software profession; in line with modern trends in other industries, it tends to remove barriers between narrow specialties — analysts who only deal in ethereal concepts, designers who only worry about structure, implementers who only write code — and to favor the emergence of a single category of generalists: *developers* in a broad sense of the term, people who are able to accompany part of a project from beginning to end.

The approach also departs from the dominant view in the current software engineering literature, which treats analysis and implementation (with design somewhere in the middle) as fundamentally different activities, susceptible to different methods, using different notations and pursuing different goals, often with the connotation that analysis and design are all that really matters, implementation being an inevitable chore. This view has historical justifications: from its infancy in the nineteen-seventies, software engineering was an attempt to put some order into the haphazard nature of program construction by teaching software people to think before they shoot. Hence the stress on early stages of software development, on the need to specify what you are going to implement. This is all justified, now as much as then. But some of the consequences of this essentially beneficial effort have gone too far, creating impedance mismatches between the different activities, and producing a strictly sequential model even though product and process quality demands seamlessness and reversibility.

With object technology we can remove the unnecessary differences between analysis, design and implementation — the necessary ones will manifest themselves clearly enough — and rehabilitate the much maligned task of implementation. It was natural for the pioneers of software engineering, when programming meant trying to solve many machine-dependent issues and explaining the result to the computer in a language that it could understand, usually low-level and sometimes inelegant, to detach themselves from these mundane aspects and stress instead the importance of studying abstract concepts from the problem domain. But we can retain these abstraction qualities without losing the link to the solution.

The secret is to make the concepts of programming, and the notations for programming, high-level enough that they can serve just as well as tools for *modeling*. This is what object technology achieves.

The following story, stolen from Roman Jakobson's *Essays on General Linguistics*, will perhaps help make the point clear:

> *In a far-away country, a missionary was scolding the natives. "You should not go around naked, showing your body like this!". One day a young girl spoke back, pointing at him: "But you, Father, you are also showing a part of your body!". "But of course", the missionary said with a dignified tone; "That is my face". The girl replied: "So you see, Father, it is really the same thing. Only, with us, everything is the face".*

So it is with object technology. With us, everything is the face.

28.8 KEY CONCEPTS COVERED IN THIS CHAPTER

- Object technology calls for a new process model, supporting seamless, reversible development.
- The unit for the sequential component of the lifecycle is the cluster, a set of logically related classes. Clusters can be arbitrarily nested.
- The lifecycle model relies on concurrent engineering: parallel development of several clusters, each permitted to rely on the specification of earlier ones.
- Object technology rehabilitates implementation.

28.9 BIBLIOGRAPHICAL NOTES

[M 1995] discusses further the topics of this chapter. It develops in detail the cluster model, and explores the consequences of the object-oriented software process on team organization, on the manager's role, and on the economics of software engineering.

[Baudoin 1996] is an extensive discussion of the lifecycle issues raised by object technology, also covering many other important topics such as project organization and the role of standards, and including several case studies.

The first presentation of the cluster model appeared in [Gindre 1989]. Another O-O lifecycle model, the *fountain model*, originally appeared in [Henderson-Sellers 1990] and is further developed in [Henderson-Sellers 1991], [Henderson-Sellers 1994]; it complements rather than contradicts the cluster model, emphasizing the need to iterate lifecycle activities.

A number of O-O analysis publications, in particular [Rumbaugh 1991] (the original text on the OMT method) and [Henderson-Sellers 1991], stress seamless development. For a detailed treatment of reversibility as well as seamlessness, see [Waldén 1995].

***

Wisdom sometimes blooms late in the season
Or half-way down the stairs.
Is it, my Lords, a crime of high treason
To trust the implementers?