29

Teaching the method

*E*nding our study of methodological issues, we turn our attention to one of the principal questions facing companies and universities that adopt object technology: how best to educate those who will have to apply it. This chapter presents teaching principles and points to common errors.

The first part of the discussion takes the view of someone who is in charge of organizing a training program in a company; the following parts take the view of a university or high school professor. All emphasize the pedagogical issues of O-O training, and so they should be relevant to you even if you are in neither of these positions — in particular if you are a trainee rather than a trainer.

29.1 INDUSTRIAL TRAINING

Let us start with a few general observations about how to teach object technology — either in public seminars or as part of an in-company training plan — to software professionals previously trained in other approaches.

Paradoxically, the trainer's task may be harder now than when object technology started to attract wide interest in the mid-eighties. It was new then to most people, and had an aura of heresy which made the audience listen. Today, no one will call security if one of the cocktail guests declares object-oriented tastes. This is the buzzword effect, which has been dubbed **mOOzak**: the omnipresence, in the computer press, of O-O this and O-O that, causing a general dilution of the concepts. The words flow so continuously from the loudspeakers — object, class, polymorphism… — as to seem familiar, but are the concepts widely understood? Often not. This puts a new burden on the trainer: convincing the trainees that they do not yet know everything, since no one can learn a subject who thinks he already knows it.

The only strategy guaranteed to overcome this problem applies the following plan:

Initial training: the "hit them twice" strategy

T1 • Take the initial training courses.

T2 • Try your hand at O-O development.

T3 • Take the initial training courses.

T3 is not a typo: after having tried to apply O-O ideas to real development, trainees take the class again. O-O training companies sometimes suggest this strategy to their customers, not always with success since it suspiciously looks like a marketing ploy to sell the same thing twice. But that is not the case.

The second iteration is what really gets the concepts through. Although the first is necessary to provide the right background, it may not be fully effective; partly because of the mOOzak effect, your students may not quite internalize the concepts. Only when they have grappled with the day-to-day challenges of object-oriented software construction — *Is a new class necessary for this concept*? *Is this a proper use of inheritance*? *Do these two features justify introducing a new node in the inheritance structure*? *Is this design pattern from the course relevant here*? — will they have the necessary preparation to listen properly. The second session will not, of course, be identical to the first (if anything, the audience's questions will be more interesting), and might straddle the border between training and consulting; but it is really a second presentation of the same basic material — not merely an advanced course following an elementary one.

In practice only the more enlightened companies are ready to accept the "teach it once, then teach it again" strategy. Others will dismiss the idea as a waste of resources. In my experience, however, the result is well worth the extra effort. The strategy is the best I know to train developers who truly understand object technology and can apply it effectively to serve the company's needs.

The next principle addresses *what* should be taught:

Training Topics principle

Especially in initial training, focus on implementation and design.

Some people assume that the curriculum should start with object-oriented analysis. This is a grave mistake. A beginner in object technology cannot understand O-O analysis (except in the mOOzak sense of recognizing the buzzwords). To master O-O analysis, you must have learned the fundamental concepts — class, contracts, information hiding, inheritance, polymorphism, dynamic binding and the like — at the level of implementation, where they are immediately applicable, and you must have used them to build a few O-O systems, initially small and then growing in size; you must have taken these projects all the way to completion. Only after such a hands-on encounter with the operational use of the method will you be equipped to understand the concepts of O-O analysis and their role in the seamless process of object-oriented software construction.

Two more principles. First, do not limit yourselves to introductory courses:

Advanced Curriculum principle

At least 50% of a training budget should be reserved for non-introductory courses.

Finally, do not consider developers alone:

Manager Training principle

A training curriculum should include courses for managers as well as software developers.

It is unrealistic, for a company or group that is adopting object technology on any scale, to hope to succeed by training developers only. Managers, regardless of the depth of their technical background, must be introduced to the basic O-O ideas and apprised of their repercussions on distribution of tasks, team organization, project lifecycle, economics of software development. The lifecycle discussion of the next chapter and, more exhaustively, management-oriented books such as [Goldberg 1995], [Baudoin 1996] and [M 1995], are typical of the material to be covered in such (usually short) courses.

> Here is an example of what manager education must include to avoid potential trouble, allow effective development and benefit the bottom line. The industry's measures of productivity are still largely based, deep-down, on ratios of produced code to production effort. In a reuse-conscious software process, you may spend some time improving software elements that *already work well* to increase their potential for reuse in future projects. This is the generalization task, an important step of the lifecycle model presented in the next chapter. Often, such efforts will *remove* code, for example because you have given a common ancestor to two originally unrelated classes, moving commonality to that ancestor. In the productivity ratio, the numerator decreases (less code) and the denominator increases (more effort)! Managers must be warned that the old measures do not tell the whole story, and that the extra effort actually improves the software assets of the company. Without such preparation, serious misunderstandings may develop, jeopardizing the success of the best planned technical strategies.

29.2 INTRODUCTORY COURSES

Let us turn our attention now to the teaching of object technology in an academic environment (although many observations will also be applicable to industrial training).

As the software community recognizes the value of the object-oriented approach, the question increasingly arises of when, where and how to include object-oriented concepts, languages and tools in a software curriculum – university, college or even high school.

Phylogeny and ontogeny

When should we start?

The earlier the better. The object-oriented method provides an excellent intellectual discipline; if you agree with its goals and techniques, there is no reason to delay bringing it to your students; you should in fact teach it as the first approach to software development. Beginning students react favorably to O-O teaching, not because it is trendy, but because the method is clear and effective.

This strategy is preferable to a more conservative one whereby you would teach an older method first, then *unteach* it in order to introduce O-O thinking. If you think object-oriented development is the right way to go, there is no reason to make a detour first.

Teachers may unconsciously tend to apply an idea that was once popular in biology: that ontogeny (the story of the individual) repeats phylogeny (the story of the species); a human embryo, at various stages of its development, vaguely looks like a frog, a pig etc. Transposed to our subject, it means that a teacher who first learned Algol, then went on to structured design and finally discovered objects may want to take his students through the same path. There is little justification for such an approach, which in elementary education would make students first learn to count in Roman numerals, only later to be introduced to more advanced "methodologies" such as Arabic numerals. If you think you know what the right approach is, teach it first.

Paving the way for other approaches

One of the reasons for recommending (without fear of fanaticism or narrow-mindedness) the use of object technology right from the start is that, because the method is so general, it prepares students for the later introduction of other paradigms such as logic and functional programming – which should be part of any software engineer's culture. If your curriculum calls for the teaching of traditional programming languages such as Fortran, Cobol or Pascal, it is also preferable to introduce these later, as knowledge of the object-oriented method will enable students to use them in a safer and more reasoned way.

O-O teaching is also good preparation for a topic which will become an ever more prevalent part of software education programs: formal approaches to software specification, construction and verification, rooted in mathematics and formal logic. The use of assertions and more generally of the Design by Contract approach is, in my experience, an effective way to raise the students' awareness of the need for a sound, systematic, implementation-independent and at least partially formal characterization of software elements. Premature exposure to the full machinery of a formal specification method such as Z or VDM may overwhelm students and cause rejection; even if this does not occur, students are unlikely to appreciate the merits of formality until they have had significant software development experience. Object-oriented software construction with Design by Contract enables students to start producing real software and at the same time to gain a gentle, progressive exposure to formal techniques.

Language choice

Using the object-oriented method for introductory courses only makes sense if you can rely on a language and an environment that fully support the paradigm, and are not encumbered by ghosts of the past. Note in particular that "hybrid" approaches, based on object-oriented extensions of older languages, are unsuitable for beginning students, since they mix O-O concepts with unrelated remnants from other methods, forcing the teacher to spend much of the time on excuses rather than concepts.

In C-based languages, for example, just explaining why an array and a pointer have to be treated as the same notion — a property having its roots in optimization techniques for older hardware architectures — would consume precious time and energy, which will not be available for teaching the concepts of software design. More generally, students

would be encouraged, at the very beginning of their training, to reason in terms of low-level mechanisms – addresses, pointers, memory, signals. They would inevitably spend much of their time, if they eventually produce a compilable program, chasing various bugs. The approach would leave the students perplexed and might end up in disaster.

An introductory course must do the reverse: present the students with a clear, coherent set of practical principles. The notation must directly support these principles, ensuring a one-to-one correspondence between method and language. Any time you spend explaining the language per se is time lost. With a good language, you explain the concepts, and use the notation as the natural way to apply them.

Although the main quality of an introductory language is its structural simplicity and its support of O-O ideas such as class-based modularization, design by contract, static typing and inheritance, you should not underestimate the role of syntactic clarity. C++ and Java texts are replete with lines such as

Examples from the basic book on Java, [*Arnold 1996*].

```
public static void main(String[] args {
if (this–>fd == –1 && !open_fd(this))
if ((xfrm = (char *)malloc(xfrm_len + 1)) == NULL) {
```

showing cryptic and confusing syntax relying on many special operators. Beginners should not be subjected to such contortions, justified only by historical considerations; learning to program well is hard enough without the interposed obstacle of a hostile notation.

Exceprts from posting of 15 October 1996.

David Clark from the University of Canberra went through this experience and posted some of his conclusions on Usenet:

> *Last semester I taught the second half of a first year programming* [course] *using Java… My experience has been that students do not find Java easy to learn. Time and again the language gets in the way of what I want to teach. Here are some examples*:
>
> - *The first thing they see is* `public static void main (String [ ] args) throws IOException` *There are about 6 different concepts in that one line which students are not yet ready to learn…*
> - *You get output for "free", but have to jump through several hoops to input anything.* (*import, declare, initialize.*). *The only way to read a number from the keyboard is to read a string and parse it. Again, this is something that crops up in the first lecture.*
> - *Java treats the primitive data types* (`int`, `char`, `boolean`, `float`, `long`,…) *differently from other objects. There are Object-type equivalents* (`Integer`, `Boolean`, `Character` *etc.*). *There is no relation between* `int` *and* `Integer`.
> - *The* `String` *class is a special case.* (*Again, for efficiency.*) *It is only used for strings that don't change. There is a* `StringBuffer` *class for strings that do change. Fair enough. but there is no relationship between* `String` *and* `StringBuffer`. *There are few features in common.*
> - *The lack of generics means that you are forever casting if you want to use a collection of elements such as* `Stack` *or* `Hashtable`. [*These things*] *are hurdles for beginning students, and distract them from the main learning outcomes of the course.*

Prof. Clark goes on to compare this experience with his practice of teaching with the notation of this book, for which, he writes, "*I do virtually no language teaching beyond giving some examples of code*".

The initial notations taught to students, so important to their future vision, must always be simple and clear, to allow in-depth understanding of the basic concepts. Even Pascal, the traditional choice of computing science departments for introductory teaching, is preferable in this respect to a hybrid language since it provides a solid, consistent basis, from which students can later move to another solid, consistent approach. It is of course even better, as noted, if the basis can be solid, consistent *and* O-O.

Some hybrid languages are industrially important; but they should be taught later, when students have mastered the basic concepts. This is not a new idea: when computing science departments adopted Pascal in the nineteen-seventies, they also included service courses to teach Fortran, Cobol or PL/I as requested by industry then. Similarly, a modern object-based curriculum may include a C++ or Java service course to satisfy downstream requirements and enable the students to include the required buzzwords on their résumés. Students will understand C++ and Java better anyway after having been taught the principles of object technology using a pure O-O language. Introductory courses, which shape a student's mind forever, must use the best technical approach.

Some teachers are tempted to use C hybrids because of perceived industry pressures. But this is inappropriate for several reasons:

- Industry demands are notoriously volatile. A few years ago, ads were all for things like RPG and Cobol. In late 1996 they were all for Java, but in 1995 no one had heard of Java. What will they list in 2010 or 2020? We do not know, but we must endow our students with capabilities that will still be marketable then. For this we must emphasize long-term design skills and intellectual principles.

- Starting with these skills and principles does not exclude teaching specific approaches later. In fact it helps, as already noted. A student who has been taught O-O concepts in depth, using an appropriate notation, will be a better C++ or Java programmer than one whose first encounter with programming involved fighting with the language.

- The historical precedent of Pascal around 1975 shows that computing science teachers can succeed with their own choices. At that time, no one in industry requested Pascal; in fact, almost no one in industry had heard of Pascal. Industry, if anything, would have requested one of the Three Tenors of the moment: Fortran, Cobol and PL/I. The computing scientists chose to go with the best technical solution, corresponding to the state of the art in programming methodology (structured programming). The result proved them right, as they were able to teach students the abstract concepts and techniques of software development while preparing them for learning new languages and tools.

29.3 OTHER COURSES

Beyond introductory courses, the object-oriented method can play a role at many stages of a software curriculum. Let us review the corresponding uses.

Terminology

The organization of higher education differs widely among countries. To avoid any confusion we must first decide on a reasonably universal terminology to denote the various levels of study. Here is some attempt at common ground:

- High school (US), lycée, Gymnasium, called secondary education below.
- First few years of university or equivalent: this is called "undergraduate studies" in the US and other Anglo-Saxon countries (*Gakubu* in Japan). In France and countries influenced by its system it corresponds to either the combination of *classes préparatoires* with the first two years of engineering schools, or to the first and second *cycles* of universities. In the German system it is the *Grundstudium*. The term "undergraduate" will be retained below.
- Finally for the later years, leading to advanced degrees, we can use the US term "graduate". (The rough equivalents are "postgraduate" in the UK; third cycle, DEA, DESS, options of engineering schools in France; *Hauptstudium* in Germany; *Daigakuin* in Japan.)

Secondary and undergraduate studies

At the secondary or undergraduate level the object-oriented method can play a central role, as noted, in an introductory programming course. It can also help for many other courses. We may distinguish here between courses that can be entirely taught in an object-oriented way, and those which will benefit from some partial use of object-oriented ideas.

Here are some of the standard courses that can be taught in a fully O-O way:

- Data structures and algorithms. Here the techniques of Design by Contract are fundamental: characterizing routines by assertions, specifying data structures with class invariants, associating loop variants and invariants with algorithms. In addition, an innovative and powerful way to organize such a course is to design it around an existing **library** of software components from an existing object-oriented environment. Then instead of starting from scratch students can learn by imitation and improvement. (More on this topic below.)
- Software engineering. The object-oriented method provides an excellent framework to introduce students to the challenges of industrial, multi-person software development, and to evaluate the benefits and limitations of project management techniques, software metrics, software economics, development environments and the other techniques which the software engineering literature discusses (in complement to object orientation) as answers to this challenge.
- Analysis and design. Clearly this can be taught in a fully O-O way; again Design by Contract is central. Courses should emphasize the seamless transition to implementation and maintenance.
- Introduction to graphics; introduction to simulation; etc.

Courses that may benefit from heavier or lighter object doses include: operating systems (where the method helps understand the notion of process, the message passing paradigm, and the importance of information hiding, clearly defined interfaces and limited communication channels in the design of proper system architectures); introduction to

formal methods (as noted above); functional programming; logic programming (where the connection with assertions should be emphasized); introduction to artificial intelligence (where inheritance is a key concept for knowledge representation); databases (which should reserve a central place for the notion of abstract data type, and include a discussion of object-oriented databases).

Even computer architecture courses are not immune from the influence of O-O ideas, as concepts of modularity, information hiding and assertions can serve to present the topic in a clear and convincing manner.

Graduate courses

At the graduate level, many O-O courses and seminars are possible, covering more advanced topics: concurrency, distributed systems, persistence, databases, formal specifications, advanced analysis and design methods, configuration management, distributed project management, program verification.

A complete curriculum

This incomplete list shows the method as being so ubiquitous that it would make sense to design an entire software curriculum around it. A few institutions have made some progress in that direction. No doubt in the years to come someone will jump and convince the management of some university to go all the way.

29.4 TOWARDS A NEW SOFTWARE PEDAGOGY

Not only does object technology affect what can be taught to students of software topics; the method also suggests new pedagogical techniques, which we will now explore.

An important note: the strategies described in the rest of this chapter are still somewhat futuristic. I believe that they must and will become prevalent for teaching software, but their full application will require an infrastructure which is not yet fully in place, in particular new textbooks and different administrative policies.

If you or your institution are not ready to apply such strategies, this does not mean that you should remove objects from your teaching. You can still, as described in the preceding sections, instill variable doses of object technology in your courses while retaining compatibility with your current way of teaching. And you should read the rest of this chapter anyway since, even if you do not follow its more radical suggestions, you might find an idea or two immediately applicable in a more conventional context.

The consumer-to-producer strategy

An O-O course on data structures and algorithms can, as noted above, be organized around a library. This idea actually has much broader applications.

A frustrating aspect of many courses is that teachers can only give introductory examples and exercises, so that students do not get to work on really interesting

applications. One can only get so much excitement out of computing the first 25 Fibonacci numbers, or replacing all occurrences of a word by another in a text, two typical exercises of elementary programming courses.

With the object-oriented method, a good O-O environment and, most importantly, good libraries, a different strategy is possible if you give students access to the libraries early in the process. In this capacity students are just reuse consumers, and use the library components as black boxes in the sense defined above; this assumes that proper techniques are available for describing component usage without showing the components' internals. Then students can start building meaningful applications early: their task is merely to combine existing components and assemble them into systems. In many respects this is a better introduction to the challenges and rewards of software development than the toy examples which have been the mainstay of most introductory courses.

Almost on day one of the course, the students will be able to produce impressive applications by reusing existing software. Their first assignment may involve writing just a few lines — enough to call a pre-built application, and yielding striking results (devised by someone else!). It is desirable, by the way, to use libraries that include graphics or other multimedia components, so as to make the outcome truly dazzling.

Later, students will be invited to go further. First they will be shown, little by little, the internals of some of the components. *Then* they will be asked to make some extensions and modifications, either in the classes themselves or in new descendants. *Finally* they will write their own classes (the step that would have come first in a traditional curriculum, but should not occur until they have had ample exposure to the work of their elders).

This learning process may be called "progressive opening of the black boxes" or, using a shorter name, the consumer-to-producer strategy. ("Outside-in" would also be an appropriate name.)

Consumer-to-producer strategy

S1 • Learn to use library classes, solely through their abstract specifications.

S2 • Learn to understand the internals of selected classes.

S3 • Learn to extend selected classes.

S4 • Learn to modify selected classes.

S5 • Learn to add your own classes.

If you like automotive comparisons, think of someone who first learns to drive, then is invited to lift the hood and study, little by little, how the engine works, then will do repairs — and, much later, design his own cars.

For this process to work, good abstraction facilities must be present, allowing a consumer to understand the essentials of a component without understanding all of it. The notion of **short form** of a class supports this idea by listing the exported features with their assertions, but hiding implementation properties. After students have seen and understood

the short form, they may selectively explore the internals of the class – again under the guidance of the instructor.

Abstraction

Most good introductory programming textbooks preach abstraction. Many in fact include the word "abstraction" in their titles. This is because the authors, being experienced software professionals and teachers, know that one cannot overcome the difficulties of large-scale software development without making constant efforts at abstraction.

Often, unfortunately, such preaching is lost on the students, who simply see it as another exhortation to "be good". You can indeed handle the small programming exercises favored by traditional teaching methods without too much abstraction effort. So why pay attention to the teacher's musings about the importance of abstraction? They will not, or so it seems, improve your Grade Point Average. Only when they have moved to larger developments would the students be in a position to benefit fully from this advice.

To preach is not the best way to teach. With the consumer-to-producer strategy, based on libraries, abstraction is not something to pontificate on: it is a practical and indispensable tool. Without abstraction, one cannot use libraries; the alternative would be to go into the source code, which is overwhelming (you would never get to do your own application) and may not be available anyway. Only through the short form with its high-level information and assertions — the library module in its abstract form — can the students take advantage of a library class.

Having become used, right from the start, to view classes through abstract interfaces, the students will much more easily apply the same principles when they start developing their own classes.

Note once again that these results are only possible in an environment supporting short forms, appropriate documentation and browsing tools, assertions, and distribution of libraries without the source.

Apprenticeship

The consumer-to-producer strategy is the application to software teaching of a time-honored technique: apprenticeship. As an apprentice you learn from the previous generation of master practitioners of your chosen craft, and once you have understood their techniques you try to do better if you can. For lack of available masters, one-on-one apprenticeship is necessarily of limited applicability; but here we do not need the masters themselves, just the results of their work, made available as reusable components.

This approach is the continuation of a trend that had already influenced the teaching of some topics in software education, such as compiler construction, before object technology became popular. In the seventies and early eighties, the typical term project for a compiler course was the writing of a compiler (or interpreter) from scratch. The front-end tasks of compiler construction, lexical analysis and parsing, require such a large effort that in practice the compiler could only be for a very small toy language. Even so, few students ever got past parsing to the really interesting parts: semantic analysis, code

generation, optimization. Then tools for lexical analysis and parsing, such as Lex and Yacc, became widely available, enabling students to spend less time on these front-end tasks. The producer-consumer strategy generalizes this change.

The inverted curriculum

The consumer-to-producer strategy has an interesting counterpart in electrical engineering, where Bernard Cohen has suggested an "inverted curriculum". Criticizing the classical progression (field theory, then circuit theory, power, device physics, control theory, digital systems, VLSI design) as "reductionist", the proponents of this approach suggest a more systems-oriented progression, which would successively cover:

- Digital systems, using VLSI and CAD.
- Feedback, concurrency, verification.
- Linear systems and control.
- Power supply and transmission, impedance matching requirements.
- Device physics and technologies, using simulation and CAD techniques.

The software education strategy suggested above is similar: rather than repeating phylogeny, start by giving students a user's view of the highest-level concepts and techniques that are actually applied in industrial environments, then, little by little, unveil the underlying principles.

A long-term policy

The consumer-to-producer strategy has an interesting variant applicable, for application-oriented courses such as operating systems, graphics, compiler construction or artificial intelligence, by professors who are in a position to define a multi-year educational plan.

The idea is to let students build a system by successive enhancement and generalization, each year's class taking over the collective product of the previous year and trying to build on it. This method has some obvious drawbacks for the first class (which collectively serves as advanceman for future generations, and will not enjoy the same reuse benefits), and I must confess I have not yet seen it applied in a systematic way. But on paper at least it is attractive. There hardly seems to be a better way of letting the students weigh the advantages and difficulties of reuse, the need for building extendible software and the challenge of improving on someone else's work. The experience will prepare them for the reality of software development in their future company, where chances are they will be asked to perform maintenance work on an existing system long before they are asked to develop a brand new system of their own.

Even if the context does not permit such a multi-year strategy, instructors should try to avoid a standard pitfall. Many undergraduate curricula include a "software engineering" course, which often devotes a key role to a software project to be carried out by the students, often in groups. Such project work is necessary, but often disappointing because of the time limitations due to its inclusion in a one-trimester or one-semester course. When

administratively possible, it is by far preferable to run such a project over an entire schoolyear, even if the total amount of allocated work is the same. Trimester projects, in particular, border on the absurd; they either stop at the analysis or design stage, or result over the last few weeks in a rush to code at any cost and using any technique that will produce a running program — often defeating the very purpose of software engineering education. You need more time, if only to let the students appreciate the depth of the issues involved in building serious software. A year-long project, whether or not it is part of a longer-term policy, favors this process. It is more difficult to fit into the typical curriculum than the standard course, but worth the fight.

29.5 AN OBJECT-ORIENTED PLAN

The idea of a long-term teaching strategy based on reuse, as well as the earlier suggestion of organizing an entire curriculum around object-oriented concepts, may lead to a more ambitious concept which goes beyond the scope of software education to encompass research and development. Although this concept will be appealing to certain institutions only, it deserves a little more thought.

Assume a university department (computing science, information systems or equivalent) in search of a long-term unifying project — the kind of project that produces better teaching, development of new courses, faculty research, sources of publication, Ph. D. theses, Master's theses, undergraduate projects, collaborations with industry and government grants. Many a now well-respected department originally "put itself on the map" through such a collective multi-year effort.

The object-oriented method provides a natural basis for such an endeavor. The focus of the work will not be compilers, interpreters and development tools (which may already be available from companies) but **libraries**. What object technology needs most to progress today is application-oriented reusable components, also called domain libraries. A good O-O environment will already provide, as noted, a set of general-purpose libraries covering such universal needs as the fundamental data structures and algorithms of computing science, graphics, user interface design, parsing. This leaves open entire application domains, from Web browsing to multimedia, from financial software to signal analysis, from computer-aided design to document processing, in which the need for quality software components is crying.

The choice of such a library development project as a unifying effort for a university department presents several advantages:

- Even though this is a long-term pursuit, partial results can start to appear early. Compilers and other tools tend to be of the all-or-nothing category: until they are reasonably complete, distributing them may damage your reputation more than it helps it. With libraries, this is not the case: just a dozen or two quality reusable classes can render tremendous services to their users, and attract favorable attention.
- Because an ambitious library is a large project, there is room for many people to contribute, from advanced undergraduates to Ph. D. candidates, researchers and

professors. This assumes of course that the application domain and the breadth of the library's coverage have been chosen judiciously so as to match the size of the available resources in people, equipment and funds.

- Talking about resources, the project may start with relatively limited means but is a prime candidate to attract the attention of funding agencies. It also offers prospects of industry funding if the application domain is of direct interest to companies.

See "APPENDIX: A HISTORY OF TAXONOMY", 24.15, page 864.

- Building good libraries is a technically exciting task, which raises new scientific challenges, so that the output of a successful project may include theses and publications, not just software. The intellectual challenges are of two kinds. First the construction of reusable components is one of the most interesting and difficult problems of software engineering, for which the method brings some help but certainly does not answer all questions. Second, any successful application library must rest on a *taxonomy* of the application domain, requiring a long-term effort at classifying the known concepts in that area. As is well known in the natural sciences (remember the discussion of the history of taxonomy), classification is the first step towards understanding. Developed for a new application area, such an effort, known as **domain analysis**, raises new and interesting problems.

- The last comment suggests the possibility of inter-disciplinary cooperation with researchers in various application domains, usually non-software.

- Cooperation should begin with people working in neighboring fields. Many universities have two groups pursuing teaching and research in software issues, one (often "*computing science*") having more of an engineering and scientific background, the other (often "*information systems*") more oriented towards business issues. Whether these groups are administratively separate or part of the same structure — both cases are common — the project may appeal to both, and provides an opportunity for collaboration.

- Finally, a successful library providing components for an important application area will be widely used and bring much visibility to its originating institution.

No doubt in the years to come a number of universities will seize on these ideas, and that the "X University Reusable Financial Components" or "Y Polytechnic Object-Oriented Text Processing Library" will (with better names than these) bring to their institutions the modern equivalent of what UCSD Pascal, Waterloo Fortran and the MIT's X Window system achieved in earlier eras for their respective sponsors.

29.6 KEY CONCEPTS STUDIED IN THIS CHAPTER

- In object-oriented training, emphasize implementation and design.

- In initial training for professionals, do not hesitate to repeat a session, with some time in-between for actual practice.

- Training in a company should include courses for managers as well as developers.

- Beginning programming courses, and many others, may take advantage of O-O techniques.
- For teaching, use a pure O-O language, clear and simple, supporting the full extent of the technology, in particular assertions.
- Courses should, as much as possible, be based on libraries of reusable components.
- The consumer-to-producer strategy (similar to "inverted curriculum" ideas), presents students with existing components, enabling them to write advanced applications right from the start, then lets students open the components, extend them, and produce new components by imitation through an apprenticeship process.
- More generally, a long-term library effort can be a unifying project for a department.

29.7 BIBLIOGRAPHICAL NOTES

The material in this chapter is derived from an article in the *Journal of Object-Oriented Programming*, of which a revised version was presented at TOOLS USA 93 and appears in the proceedings (see [M 1993c] for the two references). Further material about education and training issues appears in the book *Object Success* [M 1995], from which the term *mOOzak* is taken, as well as some observations regarding industry training.

Important articles about teaching programming using O-O concepts include [McKim 1992] and [Heliotis 1996].

The notion of inverted curriculum for education in electrical engineering is due to Bernard Cohen [Cohen 1991]. I am grateful to Warren Yates, chairman of the Electrical Engineering Department at University of Technology, Sydney, for bringing it to my attention. This chapter also benefited from discussions with many educators, including Christine Mingins, James McKim, Richard Mitchell, John Potter, Robert Switzer, Jean-Claude Boussard, Roger Rousseau, David Riley, Richard Wiener, Fiorella De Cindio, Brian Henderson-Sellers, Pete Thomas, Ray Weedon, John Kerstholt, Jacob Gore, David Rine, Naftaly Minsky, Peter Löhr, Robert Ogor, Robert Rannou.

An ongoing project is intended to produce an introductory programming book-cum-CD applying the "consumer-to-producer strategy", or "inverted curriculum" principle [M 199?]. But there are already a number of good introductory programming textbooks based on O-O ideas; they were listed in an earlier chapter, but here they are again, for convenience, without further comments: [Rist 1995], [Wiener 1996], [Gore 1996], [Wiener 1997] and [Jézéquel 1996].

The books were listed in the bibliography to chapter 2, on page 35.