Towards an Object-Oriented Curriculum

Bertrand Meyer

ISE

Santa Barbara (California)

Abstract

This proposal suggests a redesign of the teaching of programming and other software topics in universities on the basis of object-oriented principles. It argues that the new "inverted curriculum" should give a central place to libraries, and take students from the reuse consumer's role to the role of producer through a process of "progressive opening of black boxes".

This article is ISE Technical Report TR-EI-38/TE, January 1993. The present version is a draft. Revised versions were published in the Journal of Object-Oriented Programming, vol. 6, no. 2, May 1993, pages 76-81, and in TOOLS USA 1993.

Cite as Bertrand Meyer, *Towards an Object-Oriented Curriculum*, in *TOOLS 11, Technology of Object-Oriented Languages and Systems*, proceedings of 11th international TOOLS conference, Santa Barbara, August 1993, eds. Raimund Ege, Madhu Singh and B. Meyer, Prentice Hall 1993, pages 585-594.

Eiffel Software, ISE Building, 356 Storke Road, Santa Barbara CA 93117 USA, telephone 805-685-1006, fax 805-685-6869, http://www.eiffel.com, info@eiffel.com.

Towards an Object-Oriented Curriculum

1 INTRODUCTION

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.

This article proposes a coordinated approach to structuring such a curriculum, based on systematic reliance on the best aspects of the object-oriented method. It suggests a radical departure from the traditional methods of teaching programming, design and analysis: the *progressive opening of black boxes*, also known as the "inverted curriculum" and based on the systematic use of object-oriented libraries of reusable components. It also offers ideas for university departments that are in search of ambitious, multi-year federating projects.

Although the discussion will mostly address the question of academic education, some of it is also applicable to courses taught to professionals, either in public seminars or as part of an in-company training plan.

2 WHEN?

Start early

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 teach it as the first approach to software development. Beginning students react favorably to object-oriented teaching, not just 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 object-oriented thinking. If you think object-oriented development is the right way to go, there is no reason to delay.

Teachers sometimes have a (usually unconscious) tendency to apply the biologists' dictum that in human evolution 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 education, this means that a teacher who first learned Algol, then went on to structured design and finally discovered object orientation may want to take his students through the same path. (If Algol is the frog, structured design must be the pig.) There is little justification for such an approach which, transposed to elementary education, would mean that 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 orientation as the first method that students will learn 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 make it possible to use them in a safer and more reasoned way.

The object-oriented method 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. The use of assertions and more generally of the Design by Contract approach [7] 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 formality. Some recent developments in the area of object-oriented formal specification, such as Object-Z [2], may ease that transition by providing a natural bridge between the two areas.

A caveat

As you will probably have noted by yourself, the use of the object-oriented method for introductory programming, recommended above, only makes sense if you can rely on a language and an environment which fully support the paradigm, and are not encumbered by remnants from the past. In particular, "hybrid" approaches (based on object-oriented extensions of older languages) are unsuitable for beginning students.

Opening a C++ textbook at almost any page will provide ample evidence of this inadequacy. Here is a short extract from one of the best current C++ books [10], showing some typical code:

```
#defineMakeRPtr (T)
class RPtr (t) : public RPtr_base {
      Counted *ptr;
  public:
    RPtr(T) ()
       : RPtr_base()
       { }
    RPtr(T) (RPtr(T)& r)
       : RPtr_base(r)
       { }
    RPtr(T) (T *tp)
       : RPtr_base ((Counted *) T)
       { }
    ~RPtr (T) ()
```

```
    { }
  RPtr(T)& operator=(RPtr(T)& r)
    { *((RPtr_base *) this) = r; }
  RPtr(T)& operator=(T *tp)
    { *((RPtr_base *) this) =
      (Counted *) tp; }
  T& operator *()
    { assert(ptr); return *((T *) ptr); }
  operator T *()
    { return (T *) ptr; }
  int operator !()
    { return !ptr; }
};
```

With this kind of notation, it is impossible to teach the concepts. Most of the time will be spent on notation: trying to explain the use of various special symbols such as {, }, &, *, ~, #, ! and \, going through the differences between . and –>, or explaining the mysteries of bizarre constructions such as

return *((*T* *) *ptr*);

Just explaining why an array and a pointer have to be treated as the same notion – a central property of C-based languages, having its roots in obscure optimization techniques for hardware architectures of the nineteen-sixties – 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 any compilable program, chasing various damaging bugs. The approach would leave the students perplexed and might well end up in disaster.

In contrast, an introductory course must focus on the essential concepts and techniques, and present the students with a clear, coherent set of principles. The notation must directly support these principles; in fact there must be a one-to-one correspondence between the language and the method. The language must help the students, not confuse them.

So even teachers who believe in hybrid approaches should not use such a notation for introductory teaching. If your preference is indeed for hybrids, use a more conservative approach (such as Pascal, the traditional choice of most computer science departments for introductory teaching) for the first courses, and introduce the object-oriented method, with your favorite language and tools, in later courses. The initial notations taught to students use must always be simple and consistent.

3 WHERE?

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), lyce, 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 the French system it corresponds to either the combination of *classes prparatoires* 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 above, in an introductory programming course.

The method can of course help for many other courses. We may distinguish here between courses which can be entirely taught in an object-oriented way, and those which will benefit from some partial use of object-oriented ideas.

In the first category, we find the following courses (or their equivalents), which may be based on a fully object-oriented approach:

- 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 the best framework I know 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 (along with object orientation) as answers to this challenge.
- Analysis and design. Clearly this can be taught in a fully object-oriented way; again Design by Contract is central. It is essential here to avoid the disastrous pitfalls of earlier methods, which presented analysis and design as the "noble" activities of

system development, maintaining a wide gap with implementation, viewed as the low-level part. Object-oriented technology makes it possible to have a much more seamless approach, where the same concepts and notations are applied throughout the process; this is especially true in the Eiffel method (as discussed below). The teaching of analysis and design should be consistent with this view, and emphasize the seamless transition to implementation and maintenance. The work of Nerson [8, 9] and Henderson-Sellers [3] is particularly useful to help achieve this objective.

- Introduction to graphics.
- Introduction to simulation.

In the second category – undergraduate courses which may benefit from heavier or lighter object doses – we may note: 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 to 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 object-oriented 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 object-oriented courses and seminars are possible, covering all the areas to which researchers and advanced developers are currently applying their efforts: concurrency, distributed systems, persistence, databases, formal specifications, advanced analysis and design methods, configuration management, distributed project management, program verification.

Towards a complete object-oriented curriculum

This incomplete list shows the method as being so ubiquitous that it would make sense to design an entire software curriculum around it. I do not know any such complete curriculum yet, although some encouraging partial attempts are in progress (Universit de Nantes, Universit de Nice, Carleton University, University of Technology Sydney). No doubt in the years to come someone will jump and convince the management of some university to go all the way.

4 HOW?

Not only does object-orientation affect what can be taught to students of software topics; the method also suggests new pedagogical techniques. Here are a few suggestions, based on discussions with university professors as well as on my own experience.

Progressively opening the black boxes

It was mentioned above that that an object-oriented course on data structures and algorithms could be organized around a library. This idea deserves further consideration, as it may actually be applied to courses on introductory programming and many other subjects.

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 in an elementary programming course.

With the object-oriented method, a good object-oriented environment and, most importantly, good libraries, a less traditional 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.

With this technique 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 that have been the traditional mainstay of 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 produce striking results (devised by someone else!). Then they are invited to take the components making up the application and recombine them in different ways so as to produce variants of the application, or apply them to new uses.

This black-box use of pre-existing components is only the first step. As students progress, a process of **progressive opening of the black boxes** will take place. The students are encouraged to start looking into the components themselves. The teacher may wish to specify the order in which the components are to be thus examined.

Initially the purpose of this progressive opening is simply to let students understand the components, which provide models of good object-oriented designs. Then little by little the students are induced to adapt the components to new purposes – either by copying them and modifying the copies, or by using the inheritance mechanism, whose very purpose is indeed to support a combination of reuse and adaptation. In the process the need for new software elements will most likely arise, so the students will start writing their own classes; they only do so after having had extensive exposure to the best possible examples of quality object-oriented software – library classes.

For this process to work, good abstraction facilities must be present, making it possible to understand the essentials of a component without understanding all of it. The notion of **short form** of a class, as present in Eiffel [5, 6], supports this idea: a short form (which can be produced by tools of the environment) is an abstracted version of the class, revealing only the specification of the class, that is to say the properties which can be used explicitly by client classes. The short form lists the exported features with their assertions, but hides 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.

Apprenticeship

The technique of progressive opening of black boxes is the application to software teaching of time-honored technique of apprenticeship: learning from the previous generation of master practitioners of your chosen craft, and once you have understood their techniques trying 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 has influenced the teaching of some topics in software education before object orientation became widely popular. The evolution of the standard Compiler Construction course of computer science departments is a good example. In the seventies and early eighties, the typical term project for such a course was the writing of a complete compiler or interpreter from scratch. In practice, because the front-end tasks of compiler construction, lexical analysis and parsing, require significant development effort, the project could only be a compiler or interpreter for a very small toy language. Then tools for lexical analysis and parsing (such as Lex and Yacc on Unix) became widely available and started to be used more and more frequently for course projects; this made it possible to spend less time on these front-end tasks, and to include work on the more challenging aspects of compiler construction, such as code generation. The approach outlined above may be viewed as the generalization of this trend.

The inverted curriculum

The pedagogical technique of progressive opening of the black boxes has an interesting analogy in a neighboring discipline – electrical engineering. There has been much talk in recent years, in electrical engineering circles, of an educational policy known as the inverted curriculum [1]. The proponents of this approach criticize the classical electronics curriculum (field theory, then circuit theory, power, device physics, control theory, digital systems, VLSI design) as "reductionist" and suggest instead to use a more "systems-oriented" approach. In order:

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

The ideas seem 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 the most advanced industrial environments, then, little by little, unveil the underlying principles.

A long-term policy

The "progressive opening" approach has an interesting variant applicable by professors who are in a position to define a multi-year educational strategy. This variant is relevant for courses on application-oriented topics such as operating systems, graphics, compiler construction or artificial intelligence.

To teach such an application area, it is interesting to have the 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 an attractive idea. 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.

A practical note is in order here. Even if the context does not permit such a multi-year strategy, instructors in charge of software education 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 often disappointing because of the time limitations stemming from its inclusion in a one-trimester or one-semester course. If the academic administration can at all be convinced, it is much preferable to run such a project over an entire schoolyear (even 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 in a rush over the last few weeks to code at any cost and using any technique that will produce a running program – often defeating the very purpose of software engineering education. It is desirable to have a little more time on your hands, so as 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 as suggested above, favors this process. It is a little more difficult to fit into the typical curriculum than the standard trimester or semester course, but well worth the fight.

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 is worth some thought.

This discussion applies to a university department (computing science, information systems or equivalent) which is 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, PhD 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-oriented technology needs most to progress today is application libraries (also called domain libraries). With a good object-oriented environment, as already noted, will come 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 financial software to signal analysis, computer-aided design and many others – 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 such an effort 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 PhD 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, such a project may start with relatively limited means but is a prime candidate to attract the attention of funding agencies. It also offers prospects of funding by industry if the application domain is one which is of direct interest to companies.
- 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 since the work of Linnus and Buffon, 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 whose specialty is in the application domain rather than in software engineeering.
- 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.

I have no doubt that 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.

6 THE ROLE OF EIFFEL

It should be clear from the preceding discussions that Eiffel was designed as the vehicle for a new approach to teaching software, based on the ideas described above.

Here Eiffel denotes not just a programming language, but a methodology for software construction, based on a language (covering analysis and design as well as implementation and maintenance), a set of methodological principles, libraries, and tools. It is particularly important to note that the approach is not just restricted to implementation, but covers the entire lifecycle and promotes a seamless approach to system development, meant to avoid the "impedance mismatches" between successive steps promoted by traditional approaches (and, regrettably, by much of the recent work in object-oriented analysis). This makes it possible to attract and reconcile faculty and students from both of the two main sub-cultures of software engineering: computing science and information systems.

Of course Eiffel is meant for industrial development, not just teaching. But as a vehicle for teaching, it presents a number of important properties:

- A generally recognized "clean" and consistent design.
- Coverage of many aspects of software development, from analysis to implementation and maintenance.
- Inclusion of many principles of modern software engineering.
- Use of assertions, disciplined exceptions, and systematic techniques of software construction ("Design by Contract").
- Presence of carefully designed libraries covering many areas of computer science (data structures, graphics, user interfaces, language analysis), and serving as ready-made models for students, so as to support the apprenticeship process described above.
- Availability of source for the libraries, which is essential for the above ideas to be implemented successfully.
- Advanced tools, in particular in the area of graphical user interfaces.
- As noted above, a seamless approach making it possible to unite rather than separate the different steps and views of software construction.
- Existence of good textbooks (although more are needed).
- Widely available implementations, with cheap academic licenses.

Around 1975, the educational community switched worldwide to the use of Pascal as the vehicle of choice to teach computing science. This move was not the result of industry demand: if anything, the industry would have suggested Fortran, Cobol or PL/I. It was not the result of a desire to adhere to "standards": the only languages to have been reasonably standardized then were, apart from the ones just mentioned, Algol 60 and Algol 68. It was simply a realization, on the part of the academic community, that teaching must be done *right*, whatever the commercial pressures may be.

After playing its role admirably for some twenty years, Pascal is, in the mind of many educators, ready for retirement. As this article has shown, object-oriented technology, in its serious variant, is ready to take its place. It is the ambition of Eiffel to serve as the primary tool for teaching programming and software engineering in a modern, user-friendly and systematic way, from the most elementary introduction to the most advanced courses. We feel that Eiffel in its current incarnation is ready to take that role.

Acknowledgments

A short sabbatical in August-October 1992 at University of Technology, Sydney, enabled me to practice and validate some of the ideas developed here, with important contributions from John Potter and David Morgan of UTS. This article also benefited from the experience of James McKim's experience of teaching software engineering with Eiffel at the Hartford Graduate Center as reported in [4]. The feedback of other university professors who have taught various subjects using object-oriented technology and Eiffel over the years is also gratefully acknowledged; they include Jean Bzivin (U. of Nantes, France), Jean-Claude Boussard and Roger Rousseau (U. of Nice, France), Brian Henderson-Sellers (U. of New South Wales, Australia), J.H. Kerstholt (T.H. Enschede, The Netherlands), Peter Lhr (T.U. Berlin, Germany), Naftaly Minsky (Rutgers U., USA), Christine Mingins (Monash U., Australia), R. Ogor and R. Rannou (ENST, France), David Riley (U. of Wisconsin, USA), David Rine (George Mason U., USA), Robert Switzer (U. of Gttingen, Germany), Pete Thomas and Ray Weedon (Open U., Great Britain), and many others.

References

[1] B. Cohen, "The Inverted Curriculum", Report, National Economic Development Council, London., 1991.

[2] Roger Duke, Paul King, Gordon Rose and Graeme Smith, "The Object-Z Specification Language", in *TOOLS 5* (Proceedings of TOOLS USA 1991), ed. T. Korson, V. Vaishnavi and B. Meyer, pp. 465-483, Prentice Hall, Englewood Cliffs (N.J.), 1991.

[3] Brian Henderson-Sellers, *A BOOK of Object-Oriented Knowledge*, Prentice Hall Object-Oriented Series, Sydney (Australia), 1991.

[4] James McKim, "Teaching Object-Oriented Programming and Design", *Eiffel Outlook*, vol. 2, no. 3, pp. 8-19, September-October 1992. See also tutorial notes, TOOLS USA 1992, Santa Barbara (Calif.), August 1992.

[5] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[6] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1991.

[7] Bertrand Meyer, "Applying "Design by Contract"", *IEEE Computer*, vol. 40-51, no. 10, October 1992.

[8] Jean-Marc Nerson, "Applying Object-Oriented Analysis and Design", *Communications of the ACM*, vol. 35, no. 9, pp. 63-74, September 1992.

[9] Jean-Marc Nerson and Kim Walden, *Seamless Architecturing of Object-Oriented Software*, Prentice Hall International Object-Oriented Series, Hemel Hempstead, 1993. To appear.

[10] Jonathan S. Shapiro, *A C++ Toolkit*, Prentice Hall Series in Innovative Technology, 1991.