The Outside-In method of teaching introductory programming

Bertrand Meyer
ETH Zürich, Chair of Software Engineering
(Also Eiffel Software, Santa Barbara, and Monash University, Melbourne)

se.inf.ethz.ch

Abstract

The new design for the introductory programming course at ETH relies on object technology, Eiffel, extensive reuse, a graphics-rich library (TRAFFIC) built specifically for the course, a textbook ("Touch of Class"*) and an* **Outside-In** *approach based on "inverted curriculum" ideas. This article presents the key aspects of the approach.*

Note: readers interested in following the development of our course, the "Touch of Class" textbook and the supporting TRAFFIC software project may look up the page se.inf.ethz.ch/touch, where they can also subscribe to mailing lists connected with the approach.

1 The context

Many computer science departments around the world are wondering today how best to teach introductory programming. This has always been a difficult task, but new challenges have been added to the traditional ones:

- There is a strong pressure from many sources to emphasize directly operational skills over deeper, long-term concepts.
- Pressure also come from student families — more influential nowadays than in the past — who focus on the specific skills required in the job ads of the moment, and don't necessarily realize that four years later the acronyms listed in these ads might be different.
- Many academics who push fashionable technologies by invoking the demands of industry misunderstand industry's real needs: real industry recruiters — at least the good ones — know to look for problem-solving skills rather than narrow knowledge.
- Students come with a wide variety of backgrounds. Some have barely touched a computer; others may have programmed extensively before. It's tempting to assume programming experience, but this is unfair to students from the first category, who will then almost automatically fail, even though some may have developed other skills — such as mathematics — and have the potential to become good computer scientists.
- Of the students who have programming experience, some may actually possess skills (such as Web site programming) that the teacher doesn't.
- For exercises, if the programming language is C++ or Java, students increasingly resort to what may be called *Google-and-Paste* programming: find on the Web some existing program that — among other things — does the job, and turn off the irrelevant parts. It's a form of "reuse" not accounted for in the software engineering literature, and may yield solutions of 10,000 lines in which only a hundred are actually used. Unless we explicitly forbid this practice (should we?) it's not cheating, but it does raise interesting questions about our pedagogical techniques.
- Many courses use an object-oriented language, but not necessarily in an object-oriented way; few people have managed to blend genuine O-O thinking into the elementary part of the curriculum.
- In continental Europe a local phenomenon adds to the confusion: what we may call *Confetti Bolognese*, the tearing into pieces of national university systems — German, French, Italian..., each with its own logic and traditions — to make them all conform (as part of the "Bologna process") to a standard Bachelor's/Master's program, in principle devised from the US/UK model but in practice missing some of its ingredients. This is both an opportunity to take a fresh look at the curriculum and a source of instability.

The teaching of computer science at ETH, the Swiss Federal Institute of Technology in Zurich, is subjected to these contradictory pressures as in every other institution. What perhaps is most specific of ETH is its prestigious tradition of introducing new techniques both for programming and for teaching programming, epitomized in the line of languages designed by Niklaus Wirth: Pascal, Modula-2, Oberon. So it's not without trepidation that I accepted the invitation to take responsibility for "Introduction to Programming", formerly "Informatik I", starting in the Fall of 2004, and to teach it in Eiffel. Rather than using safe, time-honored approaches, I am taking the plunge and trying to apply ideas ("Progressive Opening of the Black Boxes", Inverted Curriculum, full use of object technology) which I previously described — from the comfort of a position in industry — in articles and books.

One of the remarkable traditions of computer science at ETH is the rule that introductory courses must be taught by senior professors. This contrasts with the practice of some institutions which view the teaching of introductory programming as a chore best handed over to part-timers or junior faculty. The ETH approach recognizes that those who have accumulated the most experience in their discipline are the ones who should teach it to newcomers at the most elementary level. Being asked to teach such a course is a great honor, and this article explains what I am trying to do, with my group, to fulfill that duty.

2 Components

The effort includes the following elements:

- The course itself: lecture schedule, overhead slides, exercises.
- A new textbook, "Touch of Class" [7], available to the students in electronic form for the first session, with the expectation of publishing it as an actual book in 2004. The developing manuscript is available to other interested parties from the URL listed at the beginning of this article.
- An extensive body of supporting software, called TRAFFIC, also under development.

3 Outside-In: the Inverted Curriculum

The order of topics in programming courses has traditionally been bottom-up: start with the building blocks of programs such as variables and assignment; continue with control and data structures; move on if time permits — which it often doesn't in an intro course — to principles of modular design and techniques for structuring large programs.

This approach gives the students a good practical understanding of the fabric of programs. But it may not always teach the system construction concepts that software engineers must master to be successful in professional development. Being able to produce programs is not sufficient any more today; many people who are not professional software developers can do this honorably. What distinguishes the genuine professional is a set of system skills for the development and maintenance of possibly large and complex programs, open for adaptation to new needs and for reuse of some of their components. Starting from the nuts and bolts, as in the traditional "CS1" curriculum, may not be the best way to teach these skills.

Rather than bottom-up — or top-down — the approach of the course is **Outside-In**. It relies on the assumption that the most effective way to learn software is to *use* good existing software, where "good" covers both the quality of the code — since so much learning happens through imitation of proven models — and, almost more importantly, the quality of its *interfaces*, in the sense of program interfaces (APIs).

From the outset we provide the student with powerful software: an entire set of sophisticated libraries — TRAFFIC — where the top layers have been produced specially for the course, and the basic layers on which they rely (data structures, graphics, GUI, time and date…) are existing libraries that have stood the test of extensive commercial usage for many years.

All this library code is available in source form, providing a repository of high-quality models to imitate; but in practice the only way to use them for one's own programs, especially at the beginning, is through interfaces, also known as *contract views*, which provide the essential information abstracted from the actual code. By relying on contract views, students are able right from the start to produce interesting applications, even if the part they write originally consists of just a few calls to library routines. As they progress, they learn to build more elaborate programs, and to understand the libraries from the inside: to "open up the black boxes". The hope is that at the end of the course they would be able, if needed, to produce such libraries by themselves.

This Outside-In strategy results in an "Inverted Curriculum" where the student starts as a *consumer* of reusable components and learns to become a *producer*. It does not ignore the teaching of standard low-level concepts and skills, since at the end we want students who can take care of everything a program requires, from the big picture to the lowest details. What differs is the order of concepts and particularly the emphasis on architectural skills, often neglected in the bottom-up curriculum.

The approach is intended to educate students so that they will master the key concepts of software engineering, in particular *abstraction*. In my career in industry I have repeatedly observed that the main quality that distinguishes good software developers is their ability to abstract: to separate the essential from the accessory, the durable from the temporary, the specification from the implementation. All introductory textbooks indeed preach abstraction, but it's doubtful how effective such exhortations can be when all the student knows of programming is the usual collection of small algorithmic examples. I can pontificate about abstraction as much as the next person, but the only way I know to convey the concepts effectively is by example; in particular, by showing to the student how he or she can produce impressive applications through the reuse of existing software made of tens of thousands of lines, resulting from maybe a hundred person-years of work, so that trying to understand it from the inside, by reading the source code, would take months of study. Yet the student can, in the first week of the course, produce impressive results by reusing that software through its abstract interfaces — the contract views.

Here abstraction is not just a nice idea that we ask our students to heed, another parental incitation to be good and do things right. It's the only way to survive when faced with an ambitious goal that you can't fulfill except by standing on someone else's shoulders.

The student who has gone early and often through this experience of building a powerful application through interface-based reuse of libraries does not need to be harangued much more about the benefits of abstraction and reuse. These concepts become a second nature. Teaching is better than preaching, and if something is better than teaching it must be the demonstration, carried out by the students themselves, of the principles at work, producing "Wow!" results.

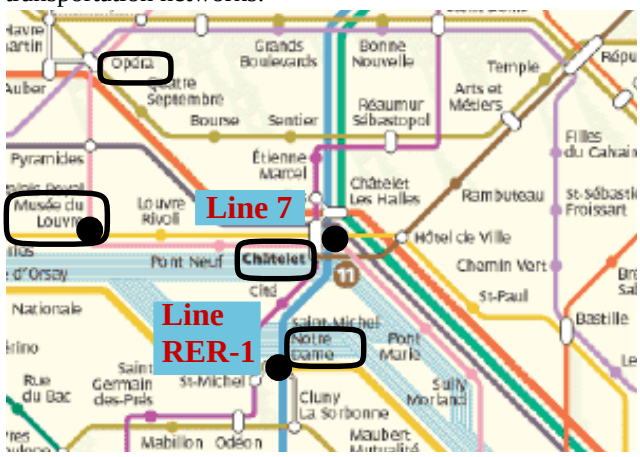
4 The supporting software

Central to the Outside-In approach of our course is the accompanying TRAFFIC software, available in source form on the book's CD and also on the Web. The choice of application area for the library required some care; the problem domain had to:

- Be immediately familiar to any student, so that we could spend our time studying software issues and solutions, not understanding the context.
- Provide a large stock of interesting algorithms and data structure examples, applications of fundamental computer science concepts, and new exercises that each instructor can devise beyond those in the *Touch of Class* textbook.
- Call for graphics and multimedia development as well as advanced Graphical User Interfaces — a requirement that is particularly important to capture the attention of a student generation which has grown up with video games and sophisticated GUIs.
- Unlike many video games, not involve violence and aggression, which would be inappropriate in a university setting (and also would not help correct the gender imbalance which plagues our field).

The application area that we retained meets these criteria. It's the general concept of *transportation in a city*: modeling, planning, simulation, display, statistics. The supporting TRAFFIC software is a library, providing reusable components from which students and instructors can build applications. Although still modest, the library has the basic elements of a Geographical Information System, and the supporting graphical display mechanisms.

Our example city is Paris, with its sights and transportation networks:



Since the city's description comes from XML files, it is possible to retarget the example to another city.

The very first application that the student produces displays a map, highlights the Paris Metro network on the map, retrieves a predefined route, and shows a visitor traveling that route through video-game-style graphical animation. The code is:

```
class PREVIEW inherit
        TOURISM
feature
        explore is
                        -- Show city info and route.
                do
                        Paris.display
                        Louvre.spotlight
                        Metro.highlight
                        Route1.animate
                end
end
```

The algorithm is four lines of code, and yet the effect is quite stunning, since these lines call powerful TRAFFIC mechanisms. In this example as in all subsequent ones, we remove any impression of "magic", since we can explain everything, at an appropriate level of abstraction. We avoid ever having to say "just code this the way you're told, it will work, you'll understand later". Such an approach is unsound pedagogically and does not create confidence in the subject. In this example, even the **inherit** clause can be explained in a simple fashion: we don't go into the theory of inheritance, of course, but simply tell the student that class *TOURISM* is a helper class introducing pre-defined objects such as *Paris*, *Louvre*, *Metro* and *Route1*, and that a new class can "inherit" from such an existing class to gain access to its features. They're also told that they don't need to look up the details of class *TOURISM*, but may do so if they feel the engineer's urge to know "how things work".

So the way we enable our students to approach the topics progressively is always to abstract and never to lie, even if it would be a pedagogical lie.

5 Object technology and model-driven architecture

Using an object-oriented language for introductory programming is not a new idea. But it's sometimes done half-heartedly, with the implicit view that students must be taken through the same set of steps that their teachers had to climb in their time. This approach continues the traditional bottom-up order of concept introduction, reaching classes and objects only as a reward to the students for having patiently climbed the *Gradus ad Parnassum* of classical programming constructs. There's no good reason for being so coy about O-O. After all, part of the pitch for the method is that it lets us build software systems as clear and natural *models* of the concepts and objects with which they deal. If it's so good, it should be good for everyone, beginners included. Or to borrow a slogan from the waiters' T-shirts at Anna's Bakery in Santa Barbara, whose coffee fueled some of the reflections behind this course: *Life is uncertain — Eat dessert first!*

Classes and objects appear indeed at the very outset, and serve as the basis for the entire course. I have found that beginners adopt object technology enthusiastically provided the concepts are introduced, without any reservations or excuses, as the normal, modern way to program.

One of the principal consequences of the central role of object technology is that the notion of *model* guides the student throughout. The emergence of "model-driven architecture" reflects the growing recognition of an idea central to object technology: that successful software development relies on the construction of models of physical and conceptual systems. Classes, objects, inheritance and the associated techniques provide an excellent basis to teach effective modeling techniques.

Object technology is not exclusive of the traditional approach. Rather, it subsumes it, much in the same way that relativity includes classical mechanics as a special case: an O-O program is made of classes, and its execution operates on objects, but the classes contain routines, and the objects contain fields on which programs may operate as they would do with traditional variables. So both the *static* architecture of programs and the *dynamic* structure of computations cover the traditional concepts. We definitely want the students to master the traditional concepts and techniques: algorithmic reasoning, variables and assignment, control structures, procedures, recursion…

6 Eiffel and Design by Contract

The approach relies on Eiffel and the EiffelStudio environment which students can download for free from www.eiffel.com. (Universities can also limit themselves to the free version, or they may get an academic license for extended functionality.) This choice directly supports the pedagogical concepts of the course:

- The Eiffel language is uncompromisingly object-oriented throughout.
- As many people have remarked, Eiffel provides an excellent basis to learn other programming languages, such as C#, Java, Smalltalk or C++. A software engineer must be multi-lingual, and in fact able to learn new languages regularly; but the first language you learn is critical since it can open or close your mind forever. Eiffel provides a general framework that is an excellent preparation for learning new paradigms later in the student's career.
- Eiffel is very easy to learn. In some other languages, before you can produce any result, you must include some magic formula which you don't understand, such as the famous *public static void main* (*string* [] *args*). This is not the case in Eiffel. The concepts can be introduced progressively, without interference between basic constructs and those not yet studied.
- The EiffelStudio development environment uses a modern, intuitive GUI, with advanced facilities including sophisticated browsing, editing, debugging, automatic documentation (HTML or otherwise), even metrics. It produces architectural diagrams automatically from the code and, the other way around, lets a user draw diagrams from which the environment will produce the code, with round-trip capabilities.
- EiffelStudio is available on many platforms including Windows and Linux.
- The environment includes a set of carefully written libraries, which support the reuse concepts of our approach, and serve as the basis of the TRAFFIC library. The most directly relevant are *EiffelBase*, which by implementing the fundamental structures of computer science directly supports the study of algorithms and data structures in part III of the textbook, *EiffelTime* for date and time, and *EiffelVision*, an advanced portable graphical library serving as the basis for the graphical parts of TRAFFIC.
- Unlike tools designed for education only, Eiffel is used commercially for large mission-critical applications handling billions of dollars of investment, managing health care, performing large civil and military simulations, and others spanning a broad range of areas. This is in my opinion essential to effective teaching of programming; a tool that is really good should be good for the professionals as well as for the novices.
- Eiffel is not just a programming language but a *method* whose primary aim — beyond expressing algorithms for the computer — is to support *thinking* about problems

and their solutions. It enables us to teach a **seamless approach** that extends across the software lifecycle, from analysis and design to implementation and maintenance. This concept of seamless development, supported by the two-way diagram tool of EiffelStudio, is in line with the key modeling benefits of object technology, and at the heart of the Eiffel approach.

To support these goals, Eiffel directly implements the concepts of **Design by Contract**™, which were developed in connection with Eiffel and are closely tied to both the method and the language. By equipping classes with preconditions, postconditions and class invariants, we let students use a much more systematic approach than is currently the norm, and prepare them to become successful professional developers able to deliver bug-free systems.

Along with these semantic concepts we shouldn't underestimate the role of *syntax*, both for experienced programmers and for beginners. Eiffel's syntax — illustrated by the above short example of the course's first program, to be compared to equivalents in other languages — seeks to facilitate learning, enhance program readability, and fight mistakes:

- The language avoids cryptic symbols.
- Every reserved word is a simple and unabbreviated English word (*INTEGER*, not *int*).
- The equal sign =, rather than doing violence to hundreds of years of mathematical tradition, means the same thing as in math. (How many students, starting with languages where = denotes assignment, have wondered what value *a* must have for *a = a + 1* to make sense, and as noted by Wirth [9] why *a = b* doesn't mean the same as *b = a*?)
- In many languages, program texts are littered with semicolons separating declarations and instructions. Most of the time there is no reason for these pockmarks; even when not consciously noticed, they affect readability. Being required in some places and illegal in others, for reasons not always clear to beginners, they are also a source of small but annoying errors. Eiffel's syntax has been designed so that the semicolon, as instruction separator, is optional, regardless of program layout. With the standard layout style of writing separate instructions and declarations on separate lines, this leads to a neat program appearance.

Encouraging such cleanliness in program texts must be part of the teacher's pedagogical goals. Eiffel includes precise style rules, explained along the way in *Touch of Class* to show students that good programming requires attention to both the high-level concepts of architecture and the low-level details of syntax and style: quality in the large and quality in the small.

More generally, a good teaching language should be unobtrusive, enabling students to devote their efforts to learning the concepts (and professional programmers to applying the concepts). This is one of the goals of using Eiffel for teaching: that students, ideally, shouldn't even have to *know* what language they are using.

7 How formal?

One of the benefits of the Design by Contract approach is to expose the students to a gentle dose of "formal" (mathematically-based) methods of software development.

The software world needs — among other advances — more use of formal methods. Any serious software curriculum should devote at least one course entirely to mathematics-based software development, for example in the second year, based on a mathematical specification language such as B, Z or HOL. In addition — although not as a substitute for such a course — the ideas should influence the entire software curriculum. This is particularly important at the introductory level. It is probably unrealistic, today at least, to subject beginners to a fully formal approach; this might turn off some of them who might otherwise become good computer scientists, and would miss the need to start mastering the concrete, practical skills of programming.

The challenge is not only to include an introduction to formal reasoning along with those practical skills, but to present the two aspects as complementary, closely related, and both indispensable. The techniques of Design by Contract, tightly woven into the fabric of object-oriented program structures, permit this.

Teaching Design by Contract awakens students to the idea of mathematics-based software development. Almost from the first examples of interface specifications, routines possess preconditions and postconditions, and classes possess invariants. These concepts are introduced in the proper context, treated — as they should, although many programmers still fear them, and most programming languages offer no support for contracts — as the normal, obvious way to reason about programs. Without intimidating students with a heavy-duty formal approach, we prepare the way for the introduction of formal methods, which they will fully appreciate when they have acquired more experience with programming.

Writing up the material has confirmed that reliance on mathematics actually helps following a practical, concrete, hands-on approach. For example we introduce loops as an *approximation* mechanism, to compute a solution on successively larger subsets of the data; then the notion of *loop invariant* comes naturally, as a key property of any kind of loop, expressing how big the approximation is at every stage. We hope that such techniques will help educate students for whom correctness concerns are a natural, ever-present component of the software construction process.

8 From programming to software engineering

Programming is at the heart of software engineering, but is not all of it. Software engineering concerns itself with the production of systems that may be large, require long-running development, undergo many changes, meet strong constraints of quality, timeliness and cost. Although the corresponding techniques are usually not taught to beginners, it's important in our view to provide at least a first introduction. Topics include debugging and testing (even with the best of modern programming methodology, this will account for a good deal of the time spent on the job, so it would be paradoxical not to teach the corresponding skills), quality in general, lifecycle models, requirements analysis (the programmers we are educating shouldn't just be techies focused on the machinery but should also be able to talk to users and understand their needs), GUI design.

9 Topics covered

The *Touch of Class* textbook, in its final form, is divided into five parts, which a course may not be able to cover entirely.

Part I introduces the basics. It defines the building blocks of programs, from objects and classes to interfaces, control structures and assignment. It puts a particular emphasis on the notion of contract, teaching students to rely on abstract yet precise descriptions of the modules they use and, as a consequence, to apply the same care to defining the interface of the modules they will produce. A chapter on "Just Enough Logic" introduces the key elements of propositional calculus and predicate calculus, both essential for the rest of the discussion. Back to programming, subsequent chapters deal with object creation and the object structure; they emphasize the modeling power of objects and the need for our object models to reflect the structure of the external systems being modeled. Assignment is introduced, together with references, only after program structuring concepts.

Part II, entitled "How things work", presents the internal perspective: basics of computer organization, programming languages, programming tools. It is an essential part of the abstraction-focused approach to make sure that students also master the *concrete* aspects of hardware and software, which define the context of system development. Programmers who focus on the low-level, machine-oriented, fine-control details are sometimes derided as "hackers" in the older sense (not the more recent one of computer vandal). There's nothing wrong with that form of hacking when it's the natural hands-on, details-oriented complement to the higher-level concepts of software architecture. Students must understand the constraints that computer technology puts on our imagination, especially orders of magnitude: how fast we can transmit data, how many objects we can store in primary and secondary memories, the ratio of access times for these two kinds.

Part III examines the fundamental "Algorithms and data structures" of computer science, from arrays and trees to sorting and some advanced examples. Here too the approach is object-oriented and library-based.

Part IV considers some more specialized object-oriented techniques such as inheritance, deferred features and constrained genericity, event-driven design, and a taste of concurrency.

Part V adds the final dimension, beyond mere programming, by introducing concepts of software engineering for large, long-term projects, with chapters on such topics as project management, requirements engineering and quality assurance.

Appendices provide an introduction to various programming languages of which the students should have a general understanding: C#, Java, C — described in some more detail since it's an important tool for accessing low-level details of the operating system and the hardware — and C++, a bridge between the C and O-O worlds.

10 Comparing with other approaches

Introductory programming education is a widely discussed issue and many techniques have been tried.

Today's commonly used textbooks tend to emphasize a particular programming language, often Java or C++. This has the advantage of practicality, and of easily produced exercises (sometimes subject to the Google-and-Paste attack cited above), but gives too much weight to the study of the chosen language, at the expense of fundamental conceptual skills. Eiffel as a language avoids making a fuss of itself; its aim, as noted, is to be unobtrusive, serving as a mere mode of expression for concepts of software architecture and implementation. That's what we teach, not the specifics of any language.

The justly famous MIT course based on Scheme [1] is strong on teaching the logical reasoning skills essential to a programmer. We definitely intend to retain this benefit, as well as the relationship to mathematics (especially in our case through Design by Contract), but feel that object technology provides students with a more concrete grasp of the issues of system construction. Not only is an O-O approach in line with the practices of the modern software industry, which has shown little interest in functional programming; more importantly for our pedagogical goals, it emphasizes **system building** skills and software architecture, which should be at the center of computer science education.

> One may also argue that the operational, imperative aspects of software development, downplayed by functional programming, are not just an implementation nuisance but a fundamental component of the discipline of programming, without which many of its intellectual challenges disappear. If so, we are not particularly helping students by protecting them from this component at the beginning of their education, presumably leaving them to their own resources when they encounter it later.

At the other extreme, one finds suggestions by Guzdial and Soloway [3] to make full use of modern technologies, such as graphics and multimedia, to capture the attention of the "*Nintendo Generation*". We retain this analysis, and include extensive graphics in the TRAFFIC software; but we strive to establish a proper balance, not letting technological dazzle push aside the teaching of timeless skills.

11 Outlook

The preparatory work described here is in a state of flux; the course has not yet been taught. In October of 2003 the first batch of ETH students will take it.

Only their reaction and, more significantly, their success in later parts of the computer science curriculum — if we can't wait for the ultimate test, their careers — will tell whether the Outside-In method can deliver on its promise of providing a modern computer science education based on reuse, object technology, Eiffel, Design by Contract and the principles of software engineering.

Acknowledgments

My understanding of how to teach programming has been shaped by discussions with numerous professional educators over the years, too numerous in fact to be listed here (see, however, the long list in chapter 29 of [5]). I should single out Bernard Cohen for his original work on the Inverted Curriculum [2], and Christine Mingins for implementing the ideas at Monash University, in a program [8] running continuously with Eiffel for almost ten years. Peter Henderson first pointed out to me the phenomenon called "Google-and-Paste programming" above. The work reported here is a collective effort of my group at ETH, involving in particular Susanne Cech, Karine Arnout, Bernd Schoeller and Michela Pedroni; the design and implementation of TRAFFIC is the work of Patrick Schoenbach and Till Bay with the support of the rest of the team and critical help from Julian Rogers and Ian King at Eiffel Software. I am grateful to other ETH professors, especially Jürg Gutknecht and Hans Hinterberger, for advice and numerous discussions, and to the FILEP project of ETH for financially supporting the development of the course and the software.

A preliminary version of this article appeared in the proceedings of the Japanese Symposium on Object Orientation (IPSJ-SIGSE), edited by Mikio Aoyama, Tokyo, August 2003.

Bibliography

[1] Harold Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, 1996.

[2] Bernard Cohen: *The Inverted Curriculum*, Report, National Economic Development Council, London, 1991.

[3] Mark Guzdial and Elliot Soloway: *Teaching the Nintendo Generation to Program*, in *Communications of the ACM*, vol. 45, no. 4, April 2002, pages 17-21.

[4] Bertrand Meyer, *Towards an Object-Oriented Curriculum*, in *Journal of Object-Oriented Programming*, vol. 6, no. 2, May 1993, pages 76-81. Revised version in *TOOLS 11* (*Technology of Object-Oriented Languages and Systems*), eds. R. Ege, M. Singh and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 585-594.

[5] Bertrand Meyer, *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997, especially chapter 29, "*Teaching the Method*".

[6] Bertrand Meyer, *Software Engineering in the Academy*, in Computer (IEEE), vol. 34, no. 5, May 2001, pages 28-35.

[7] Bertrand Meyer, *Touch of Class: Learning to Program Well — With object technology, Design by Contract, and steps to software engineering*, to be published, draft versions currently available from se.inf.ethz.ch/touch

[8] Christine Mingins, Jan Miller, Martin Dick, Margot Postema: *How We Teach Software Engineering*, in *Journal of Object-Oriented Programming* (JOOP), vol. 11, no. 9, 1999, pages 64-66, 74.

[9] Niklaus Wirth: *Computer Science Education: The Road Not Taken*, opening address at ITiCSE conference, Aarhus, Denmark, June 2002, available (September 2003) at www.inr.ac.ru/~info21/greetings/wirth_doklad_eng.htm.