Touch of Class:
Teaching introductory programming outside-in

Bertrand Meyer, ETH Zurich and Eiffel Software
http://se.inf.ethz.ch — http://www.eiffel.com

The high rate of change in information technology forces educators regularly to reconsider the way they teach programming, including at the introductory level. When I was unexpectedly asked to devise a new, Eiffel-based program for teaching introductory programming at ETH Zurich, I tried to address the new challenges that await our students when they graduate, and applied a number of pedagogical ideas which had been proposed and refined over the years [3] from the comfort of a position in industry: component-based teaching, "inverted curriculum", outside-in progression from consumer to producer, careful mix of pragmatics and mathematically-based methods, systematic use of Design by Contract, thoroughly object-oriented approach. The course relies on a new textbook, *Touch of Class* [7], and on an advanced software system, the TRAFFIC library, built specifically to support it.

The course has now been taught for the first time and we are preparing the subsequent iterations. Previous presentations of this effort [6] described it in the future; now we are in a position to draw some lessons from the first experience.

Issues in teaching introductory programming

Introductory teaching is a hot issue in many computer science or information technology departments around the world. Educators face a variety of challenges:

- *Providing students with valuable skills*. It is not enough to know how to program; many people, including from low-wage countries, can do this. Teaching just the basic technology is not rendering a good service to the students, even if (based on the job ads they see in the newspapers, and the specific buzzwords these ads mention) that's what they and their families demand. We need to train system thinkers, architects, high-level designers. Otherwise we are just giving them short-term skills that won't be of much use.
- *Overcoming fashion*. In our hype-driven field, it can be difficult to teach what we think is right if it isn't the craze of the moment.
- *Giving a system view*. The traditional programming exercise — say computing how to give back change using nickels, dimes and quarters — doesn't really address the kind of system-level skills that is needed in real software development applied to large systems, maintained by many people over a long time, facing frequent changes, and dealing with a wide variety of user needs and external constraints.
- *Balancing pragmatics and mathematics*. There's a growing acceptance that we need to teach the formal basis of programming, but this is often relegated to theory courses with little connection to the actual programming courses, still taught using an informal, theory-free approach. An unsolicited email from a Spanish student [8] commenting on an earlier book [4], makes this point very well:

> *I had to study a lot of theory of formal methods, loop invariants, algebraic specifications of ADTs and so on, but I did not have to use it when programming, only with "pen and paper". None of my teachers, in my laboratory courses, asked me about those nasty things, so I thought they were not very useful. Now I know this is false. Design by Contract is a powerful tool and the fact is it makes easy and funny using formal methods in software engineering.*

- *Dealing with the wide variety of student backgrounds*. Some have written a compiler or a Web service; others have barely touched a computer; most are somewhere in between. How do we keep the course interesting for that wide spectrum?
- *Glamour*. Students of the "Nintendo Generation" [1] expect modern, lively examples with the latest gizmos: graphics, sound, animation, Internet abilities … We must catch their interest without sacrificing the scientific value of what we teach.
- *Teaching principles without preaching*. Good software construction is all about abstraction. How do we concretely convey the benefits of abstraction and the associated (information hiding, distinction between specification and implementation, abstract data types …)? Pontificating doesn't work; students must see for themselves that abstraction is a lifeline when you're confronted with large problems or large existing programs. But in a university context students typically do not get to work with a large program.

Our curriculum design tries to address all these issues through a consistent plan.

Outside-In: the Inverted Curriculum

The approach 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 program *interfaces* (APIs).

From the outset we provide the student with powerful software: an entire set of sophisticated libraries written in Eiffel, TRAFFIC, where the top layers have been produced specially for the course, and the basic layers on which they rely—EiffelBase for data structures, EiffelVision for graphics and GUI, EiffelTime for time and date… — are widely used in large commercial applications. 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 in Eiffel 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" [3] 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. We do in fact have a lecture devoted to building a program completely from scratch. What differs is the

order of concepts and particularly the emphasis on architectural skills, often neglected in a bottom-up curriculum.

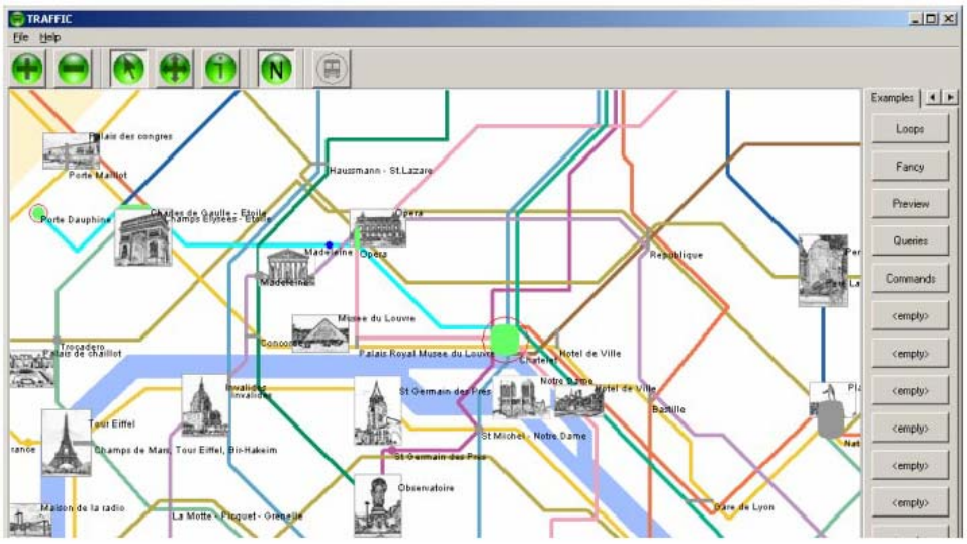
The supporting software

Central to the Outside-In approach is the accompanying TRAFFIC software, available in source form. The choice of application area for the library required some care; the domain had to:

- Be immediately familiar to any student, so that we can 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 book.
- 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 s not just an "application", doing one particular job, but a *library*, providing reusable components from which students and instructors can build applications.

The very first application that the student produces displays a map, highlights the network of the Paris Metro (or Zurich trams, or any other that the students want to use — it's all parameterizable through an XML file), retrieves a predefined route, and shows a visitor travelling that route through video-game-style graphical animation:



The algorithm takes up four lines of code, and yet its effect is impressive thanks to the underlying TRAFFIC mechanisms:

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

First lessons

The approach is very different from earlier ETH courses. The student's evaluations show that it has been well received. (We put all the evaluations and every single student comment on the course Web site [2].) In particular, I feel that we have been able to catch and retain the interest of students who were already familiar with programming as well as total beginners, not to forget those who are taking the class again as they had failed the first year exam.

Students did appreciate the use of Eiffel and the gently formal slant made possible by Design by Contract. For example the notion of loop invariant is introduced the very first time loops are mentioned (long after the notions of interface, contract and class which make up the bulk of the course). The exercise sessions have been a key complement to the lectures, helping students grasp the practical use of the concepts. About 6 weeks into the course, it appeared that some students, beginners in particular, had trouble building a program from scratch; a lecture entirely devoted to that topic corrected the problem. Only the experience of later semesters (when students get to apply the concepts) will show how successful we have been in the end, but so far all the indicators are in that the combination of reuse, contracts, object technology, Eiffel, a powerful graphical library, and the supporting textbook [7] provides for an effective introductory programming experience.

Bibliography

[1] 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.

[2] Informatik I course: course page at se.inf.ethz.ch/teaching/ws2003/37-001/.

[3] 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.

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

[5] Bertrand Meyer, *Software Engineering in the Academy*, in Computer (IEEE), vol. 34, no. 5, May 2001, pages 28-35.
[6] Bertrand Meyer, *The Outside-In Method of Teaching Introductory Programming*, in *Perspective of System Informatics*, Proceedings of fifth Andrei Ershov Conference, Novosibirsk, 9-12 july 2003, ed. Alexandr Zamulin, Lecture Notes in Computer Science 2890, Springer-Verlag, 2003.
[7] Bertrand Meyer: *Touch of Class: Learning to Program Well, through Object Technology, Eiffel, Reuse and Design by Contract*, to appear, draft available (as well as other information about the approach) at se.inf.ethz.ch/touch.
[8] José Maria Vegas Gertrudix, email to the author, 19 May 2004.