The Inverted Curriculum in Practice

Michela Pedroni
Department of Computer Science
ETH Zürich (Swiss Federal Institute of Technology)
8092 Zürich, Switzerland
michela.pedroni@inf.ethz.ch

Bertrand Meyer
Department of Computer Science
ETH Zürich (Swiss Federal Institute of Technology)
8092 Zürich, Switzerland
Bertrand.Meyer@inf.ethz.ch

Cite as follows: Michela Pedroni, Bertrand Meyer, *The Inverted Curriculum in Practice*, to appear in Proceedings of SIGCSE 2006, ACM, Houston, Texas, 1-5 March 2006.

ABSTRACT
Teaching introductory programming today presents considerable challenges, which traditional techniques do not properly address. Students start with a wide variety of backgrounds and prior computing experience; to retain their attention it is useful to provide graphical interfaces at the level set by video games; and with the ever-increasing presence of computing in society the stakes are higher, requiring a computing curriculum to introduce students early to the issues of large systems. We address these challenges through an "outside-in" approach, or "inverted curriculum", which emphasizes the reuse of existing components in an example domain involving graphics and multimedia, a gentle introduction to formal reasoning thanks to Design by Contract techniques, and an object-oriented method throughout. The new course has now been taught twice, with considerable gathering of student data and feedback; we report on this experience and its continuation.

Categories and Subject Descriptors
D.1.5 [**Programming Techniques**]: Object-oriented programming; K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms
Design, Human Factors

Keywords
Inverted Curriculum, Objects-First, Pedagogy, CS1

1. INTRODUCTION
Over the past three years we have redesigned the ETH first-year "Introduction to Programming" computer science course based on novel ideas taking their root in "Inverted Curriculum" principles [5]. The course relies on object-oriented, component-based technology with Design by Contract, and on the reuse of a large software framework built specifically for this project. It follows an "outside-in" approach where software construction relies, right from the start, on components. Students first discover the components as *consumers*, through abstract interfaces and contracts, before moving on to the *producer* perspective by exploring and modifying the implementations.

To confront these principles with the results, and to make sure we adapt what doesn't work, we carefully track students' reactions, opinions and performance.

The approach is supported by extensive material available online from the course page [14]: a textbook in progress [4]; course slides, exercises and other materials; and video recordings of all the lectures.

The new course has now been taught twice, providing enough feedback to ascertain how it is working and draw lessons for the future. We are currently teaching the third iteration. This paper presents the principles and reports on the actual results.

Section 2 explains the challenges of teaching introductory programming, which led to the design of our course. Section 3 describes this design. Section 4 introduces the concrete context and setup of the course. Section 5 reports on the student evaluations and feedback. Section 6 presents our conclusion and discusses future work.

2. TEACHING INTRODUCTORY PROGRAMMING
Teaching introductory programming today presents such challenges that it is tempting to hijack the title of Dijkstra's article "On the Cruelty of Really Teaching Computing Science" [2] to highlight a different form of cruelty: on teachers.

First, the stakes are getting ever higher. Globalization has led to massive outsourcing, with the result that those of us educating future software professionals in the industrialized world have a responsibility to teach them durable skills. It is not enough to present immediately applicable technology, for which a cheaper programmer will always be available elsewhere. This is sometimes difficult to explain to a constituency that tends to judge from current job ads with their focus on specific technical skills, but is our essential responsibility to the students.

Regardless of these economic and political aspects, programming today is no longer a rare, specialized skill but, in an elementary form, increasingly one of the "four R's". A large proportion of the population gets exposed to computers, software, and some rudimentary form of programming, for example through spreadsheet macros or Javascript for Web pages. This raises the second issue: defining precisely what we should teach to a future

professional. The ACM Curricula [9, 10] are helpful by clearly specifying different types of programming education.

This growing presence of software in non-computer-science endeavors leads to the third issue: the wide diversity of student backgrounds. Our students cover the full spectrum, from some who have barely touched a computer to those with extensive programming experience, to the point of having written an e-commerce site before they reach our first-year course. Section 5.2 gives a more precise view of the range of prior knowledge, based on student surveys. What should the teacher do in the face of such diversity? It is tempting to teach to the most advanced students only, by assuming a fair amount of experience; but this shuts out some who have the potential of becoming excellent computer scientists, and simply haven't had the opportunity or inclination to work with computers yet. We should not either — at the other extreme — bring everyone down to the lowest level; we must also find a way to catch and keep the attention of the more experienced students. The use of components, as detailed below, is a major part of our solution to this issue. By giving students access to high-quality software libraries, we let the novices take advantage of the functionality through their abstract interfaces, without needing to understand what's inside. The more advanced and intellectually curious ones can go inside the components, understand how they work, use them as guidance for their own goals, and eventually modify them.

The matter of maintaining students' attention brings up the fourth issue: quality of examples. The "Nintendo generation" [3] is unlikely to be very impressed with the small, purely abstract problems traditionally used for introductory programming. This means that while we use —in the terminology of the ACM Computing Curricula 2001 [10] — an *objects-first* approach (rather than "functional-first" or "imperative-first"), we go beyond "experimenting with [the notions of object and inheritance] in the context of simple interactive programs". Students expect more than small programs which, after all, any competent high-school student can learn to put together. Our approach is based on Traffic [13], a large software library, which provides advanced graphics, multimedia and interaction capabilities, intended to reach the quality level of today's video games and animations.

These observations are closely related to the fifth issue: how to teach the real challenges of professional software development. At university level, at least in a computer science or software engineering program, we can't just teach programming in the small. We have to prepare students for what professionals really handle: large programs. Techniques that work well for programming in the small are not sufficient in such contexts. How do we introduce students to the actual challenges of today's industrial software? The usual answer [9] is that teaching must be combined with practice, and that some issues only register when students have had more experience. While this observation is correct, it cannot be the full answer; even in a university context we need to expose students as early as possible to large programs. We address this challenge by confronting the students, from the start, with a large amount of software — much larger at least than anything that's commonly used in introductory courses: the Traffic library which (with the supporting libraries, EiffelBase, Gobo, EiffelVision and EiffelMedia) approaches the 150,000-line, 750-classes threshold.

Without the proper apparatus and method, beginning students would drown in such an abundance of software. Modern techniques of information hiding, data abstraction and especially Design by Contract are essential here. This enables us both to raise and answer the sixth issue: how do we introduce advanced but essential principles of methodology without disconnecting from the students? Such advice — to use abstraction, contracts and software principles in general — can sound preachy and unnecessary to them. Paradoxically, those who have already programmed a bit and stand to benefit most from these admonitions might be tempted to discard them since they know from experience that it is somehow possible — on small programs! — to reach an acceptable result without strict rules. Rather than preaching, the best technique is to show that a methodological principle such as the reliance on abstract interfaces with contracts makes it possible to do something that would otherwise be unthinkable: master the use of large amounts of reusable software performing sophisticated and impressive tasks, such as advanced graphics and animation. If an idea has saved you from drowning, you won't discard it as empty theoretical advice.

There are more issues, such as how to avoid "Google-and-Paste programming" [7], but the ones cited are already enough as background for the design of the course discussed here.

3. COURSE PRINCIPLES

We now describe the principles underlying our course. They are not specific to our environment; indeed a number of other institutions have applied our ideas. ETH-specific elements and the practical course setup are discussed separately in section 4.

3.1 Objects first

We use object-oriented concepts right from the start. Object technology is based on the view that software development is modeling systems. This is a natural, eminently teachable approach, especially at the introductory level where one can introduce classes that directly reflect things and concepts familiar to the students — not just objects in the material sense of the term, but also abstract notions such as "itinerary" (in a public transportation application).

Object-oriented programming has largely captured the mindset of the industry today, partly because it is both suitable for advanced applications (it seems to be the only known approach that really scales up) and based on easily understandable elementary concepts. There is no reason to deprive our students from the best known techniques and practices.

3.2 Components

As noted, we emphasize reuse from the start by giving students access to a large library, Traffic, and applications relying on Traffic — such as Flat Hunt described in section 3.6 —, all built for our course on the basis of other Eiffel libraries.

Using components has numerous advantages. It enables students to produce impressive applications from the start by relying on the power of libraries — even if initially these applications are really 10-line programs calling existing mechanisms. This takes care of the issue of catching and retaining students' interest. Most importantly, it enables us to ingrain key principles of reuse and abstraction into students' minds right from the beginning, teaching them that it's good to rely on solid existing solutions.

3.3 Abstraction and contracts

For components to be "solid" requires that they come with clear specifications. Eiffel's **contracts** — specification elements associated with software elements: routine preconditions and postconditions, class invariants [6] — fill this role, together with the associated abstraction techniques. They make the "outside-in" approach possible: beginning students can quickly learn to be successful "consumers" of components through reading their *contract forms* [6]: abstract interfaces, including contracts, extracted automatically from the software.

3.4 Order of topics

It is of course essential that students master all the traditional building blocks: variables, assignment, control structures and such. Where we differ from most existing curricula is in the order of exposition. In line with the outside-in approach, we start with what Maurice Wilkes called [11] the "outer" structure of the programming language: in our case, class interfaces, objects, features. We then progressively move to the "inner" structure.

3.5 Formality

Students need to understand that programming has a strong mathematical foundation, but here too preaching is not effective and some tact is required. We have seen curricula where students first spend two semesters studying a formal theory of software before being permitted to approach a keyboard. Since they have keyboards anyway, this risks creating a definitive gap in their minds between theory and practice, paradoxically reinforcing "hacking" attitudes that formal methods are supposed to fight in the first place. Another well-known approach is to use a functional language such as Scheme [1], or a logic language, distant from the techniques used in industry, to emphasize the mathematical basis of programming. Here too the risk exists that when students move to industry they will throw away what they have learned, finding it irrelevant. We prefer to justify a certain degree of formality — mathematically-based techniques for constructing programs — in a practical context. This makes it possible for students to write realistic programs, and to show them how these techniques, rather than being just a theoretical pet peeve of the professor, help them get these programs right. In particular, we introduce loops as an approximation technique, with the notion of loop invariant and loop variant as an integral part of the concept from the beginning. Eiffel's loop construct with its **variant** and **invariant** clauses, along with the rest of its Design by Contract mechanism, helps in this unobtrusive introduction of partially formal techniques.

3.6 The software framework

Our course fundamentally relies, as noted, on a software framework. The criteria for choosing an application domain included the following:

- It must be something with which students are immediately familiar.
- It must provide a rich base for complex algorithms and data structures, and an open-ended source of examples and exercises. In particular we are increasingly coordinating with other courses such as "Data Structures and Algorithms" and need to provide them with continuing material.
- It should include multi-media and advanced graphics.

We chose traffic in a city, with almost endless potential for modeling interesting concepts both simple and advanced, for graphics, animation, simulation, algorithms, exercises, use in other courses (for example "Data Structures and Algorithms") and extensions.

One such extension is games (which, in a university context, must be non-violent). We built one: Flat Hunt, a kind of "Scotland Yard" transposed to represent students chasing a real estate agent who only wants to rent to more respectable customers (Figure 1). Students projects produced many more.

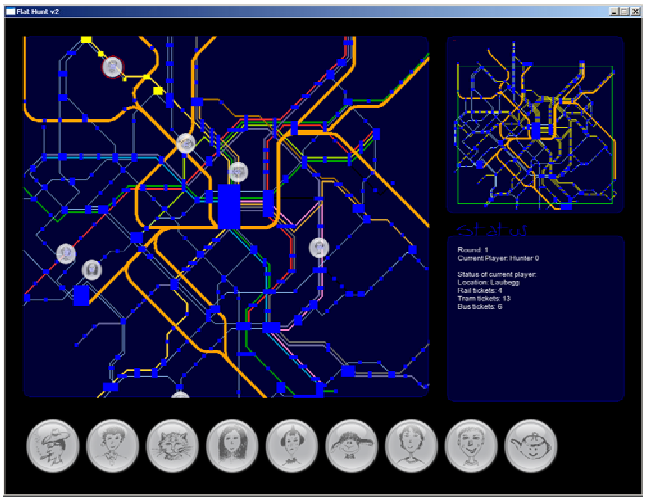

Figure 1 A Flat Hunt screenshot

The requirements for the software itself are:

- It must be very well designed and implemented, with non-cryptic interfaces (GUI and API) and documentation. The framework is intended to be the primary model and reference for the students, in their process of learning by imitation; it must be impeccable in both the large and the small.
- It must provide multiple layers of abstraction of the domain to be used at different stages in the course.

For us, this has meant a software project of a size more commonly found in industry (although not always with the same quality requirements) than universities. Thanks to a grant from the ETH education development office, complemented by a Microsoft Curriculum grant, both gratefully acknowledged, we have been able to bring this project to an acceptable stage. The result, although still not ideal, is getting close to a level where we will turn it into a public open-source project to which we hope many universities will contribute.

4. PRACTICAL SETUP

We now describe the specifics of the course organization. The first iteration of the new course was conducted in the winter semester 2003-2004 with approximately 250 students and the second the following year with about 180 students.

4.1 Course setup

The participants are future computer science graduates on their way to a bachelor's and (preferably) a master's degrees. The course is held in the first semester of the program and is the only computer science course at that point. In the ETH tradition of

providing a strong general science and engineering education to all students, the other courses are on logic, linear algebra, analysis, probability and statistics.

The weekly schedule includes four (two times two) plenary lectures by the professor and three exercise lessons by graduate and doctoral student tutors, with a group size of about 25. The duration of each lecture or lesson is 45 minutes.

The students are handed out weekly assignments from week 1 to week 9 including up to two sit-in assignments (simulating the exam). From week 10 to week 14 (semester end), students work in teams of three on a programming project.

4.2 Student body

Students fill in a questionnaire describing their prior computer and programming knowledge. The outcome (see the table in 5.2 below) confirms the diversity of the students. 22% in the first session and 14% in the second session started their computer science study without any programming experience. The percentage that did know some O-O programming before starting the course increased significantly between the two sessions from 35% to 44%. Correspondingly, the group of students that have worked with programs of more than 100 classes — a sizable experience for supposed novices — grew from 5% to 10%.

This trend has continued in the current third session (2005-2006) and seems indicative of a more general phenomenon: that "after the Internet bubble burst" we get proportionally more students attracted to computer science by genuine interest.

4.3 Course material

A new introductory programming textbook, "*Touch of Class*" [4], directly supports the course, most of the lectures being close to some of the material from one of its chapters. The textbook is in progress and currently available on line. In the first session, many chapters were being written as the course progressed, making it often uncomfortable for students. At present most of the material actually covered is present in the text.

All slides used in class are available on the Web, as well as exercises and other material. The Traffic software and the free EiffelStudio environment are also available for download. In addition, all lectures are recorded on video and put on the Web shortly after being presented. Students greatly appreciate the possibility of going over the material again at home.

4.4 Grading and exercises

There is no grading during the course, only the requirement of doing the homework, "classroom exercises" (mock exams) and project — not necessarily successfully, but showing effort — to get a certificate allowing participation in the exam, held after the year. We do correct assignments and provide constant feedback to the students; this simply has no effect on their final grades.

To the professor in charge (BM), coming from the US system, this ETH rule was initially a shock. In fact it has turned out to be tremendously helpful, enabling us to do our best teaching job without constant student obsession on grades. The course is in fact highly selective (in the terminology of [9] it is "filter" as much as "funnel", if only because of the absence of an entrance exam), but students can take a reasoned, long-term approach to learning programming.

5. STUDENT FEEDBACK

5.1 Nature of data and collection method

We systematically track students' performance and gather their assessments of our own performance. The experience of the first session led us to new ideas of things to ask about the second time around. Here are some of the elements we collect.

Initial questionnaires ask students, at the beginning of the semester, to describe their previous experience, in particular any prior exposure to programming and programming languages.

Assignment questionnaires accompany every assignment, and must be filled for the assignment to be accepted. They include questions about the assignment itself (how difficult, how useful, what was hardest, time spent etc.). They help us "feel the temperature" of students and enable tutors to give feedback and support on specific topics.

Official *end-of-semester course evaluations* conducted by the ETH administration and the Department of Computer Science cover general student satisfaction with the course, difficulty of the course, any cross-cuttings with other courses etc.

In addition, an ongoing study by Marie-Hélène Ng Cheong Vee from the University of London, applied both to our course and a similar course there, tracks actual student performance in the programming exercises by recording errors and solution paths thanks to an option of the Eiffel compiler [8].

5.2 Initial student knowledge

Tables 1 and 2 summarize some results of the initial questionnaire. The higher percentage of students with prior programming knowledge in the second session — in particular the percentage of students that have worked with large programming projects — is also reflected in a higher average of years of using computers. The same applies to the percentage of students having worked in a job where programming was a substantial part (24% in 2003/04 and 32% in 2004/05). As noted this is part of a clear trend, continuing in 2005/06.

			2003/04	**2004/05**
No programming experience			22%	14%
Some experience	No object-orientation		38%	33%
	Some O-O	Small projects	35%	43%
		Large projects (>= 100 classes)	5%	10%

Table 1 Programming experience

	2003/04	**2004/05**
One year and less	1%	1%
Two to four years	6%	1%
Five to nine years	55%	35%
Ten years and more	38%	63%

Table 2 Computer usage

5.3 Overall student satisfaction

An important indicator for the quality of the method is student satisfaction. The official evaluation of the course shows that it was very successful with the students. The average grade 4.0 (out of 5) reached in winter semester 2003/04 was even slightly improved in 2004/05 with an average grade of 4.1. This is toward

the top of student course evaluations for first- and second-year courses in the Computer Science department and is significantly higher than the grade obtained by previous versions of the course.

5.4 Satisfaction with the software

The software used for the course — Traffic in the first session, Flat Hunt running on top of Traffic the next year — gets a significantly lower appreciation than the course as a whole, although it improved from 2.7 to 2.9 in the second iteration.

Student satisfaction with the software must reach the same level as for the rest of the course, if only because Traffic is at the center of the approach. The results are, however, not hard to explain: Traffic is a major software project of the kind not commonly attempted in universities. The first version was a proof of concept. It is not surprising that it did not meet the students' expectations. We have chosen to emphasize graphics and animation because it's the best way to capture the interest of students who have grown up with video games. But for that very reason their expectations are high.

The students' overall experience is highly positive as evidenced by the final grade. Clearly Traffic is a key part of that appreciation even if the students, irritated by some aspects of Traffic, give the product itself a still insufficient grade.

6. CONCLUSION AND FUTURE WORK

The course *Introduction to Programming* is so far a success. The *Inverted Curriculum* seems to be appreciated by the students, in particular the ability to work on “real” applications right from the start.

Much work remains; the development of a system architecture and class interfaces of professional quality, providing advanced functionality, and yet simple enough to be approachable by total beginners, is a constant balancing act. The next sessions will benefit from the following changes and improvements:

- 3D graphics and sound; the newer versions of Traffic use EiffelMedia [12], a powerful multimedia library developed in our group.
- Simplified class interfaces and system architecture, through the division into several layers of functionality.

Additionally, the evaluation process needs to be further improved to provide us with more feedback on the approach:

- As the students having taken the new course move into higher semesters, we continue to track them. They will fill a questionnaire about their retrospective impressions of the approach; and we are coordinating with our colleagues teaching downstream courses to assess these students’ programming performance.
- We will develop an end-of-semester questionnaire to let us better correlate initial programming knowledge with satisfaction with the approach.
- We intend to develop a method of comparing the Inverted Curriculum approach to other approaches.

7. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 1996.

[2] E. W. Dijkstra. On the cruelty of really teaching computing science. December 1988.

[3] M. Guzdial and E. Soloway. Teaching the nintendo generation to program. *Commun. ACM*, 45(4), 2002.

[4] B. Meyer. *Touch of class: Learning to program well with object technology and design by contract*. To be published, draft versions currently available from se.ethz.ch/touch.

[5] B. Meyer. Towards an object-oriented curriculum. *Journal of Object-Oriented Programming*, 6(2):76–81, May 1993. Revised version in TOOLS 11 (Technology of Object-Oriented Languages and Systems), eds. R. Edge, M. Singh and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 585-594.

[6] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[7] B. Meyer. The outside-in method of teaching introductory programming. In Manfred Broy and Alexandre V. Zamulin, eds., *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 66–78. Springer, 2003.

[8] M. Ng Cheong Vee, B. Meyer, K.L. Mannock. Empirical study on novice errors and error paths. Available at se.ethz.ch/~meyer/publications/teaching/novices.pdf.

[9] The Joint Task Force for Computing Curricula 2005. Computing curricula 2005 (draft). April 4 2005. Available online at: www.acm.org/education/Draft_5-23-051.pdf.

[10] The Joint Task Force on Computing Curricula. Computing curricula 2001 (final report). December 2001. Available at: www.acm.org/sigcse/cc2001.

[11] M. V. Wilkes. The outer and inner syntax of a programming language. *The Computer Journal*, 11(3):260–263, November 1968.

[12] EiffelMedia project page: eiffelmedia.origo.ethz.ch.

[13] Traffic project page: se.ethz.ch/traffic.

[14] ETH “Introduction to Programming” course page: se.ethz.ch/teaching/ws2005/0001/english_index.html.