Touch of Class

The following pages reproduce the book's two prefaces (for students, for instructors).

Dedication

This book is dedicated to two pioneers of computer science, as thanks for their unending influence and in recognition of their many brilliant insights:

C.A.R. Hoare, on the occasion of his 75th birthday.

Niklaus Wirth, with special gratitude for his development of computing science (informatics) at ETH.

Prefaces

```
note
    description:"[
        This book has two prefaces, one for instructors and one for students, as stated
        here through a contrived but correct use of its own programming notation.
                ]"
class PREFACING inherit
    KIND_OF_READER
create
    choose
feature -- Initialization
    choose
            -- Get the preface that's right for you.
        do
            if is_student then
                student_preface.read
            elseif is_instructor then
                instructor_preface.read
            else
                pick_one_or_both
            end

                check
                    -- You learn about dynamic binding
                note
                    why: "You will express this more elegantly"
                end
        end
end
```

Student_preface*

*The preface for instructors is on page xxiii.

Programming is fun. Where else can you spend your days devising machines of your own imagination, build them without ever touching a hammer or staining your clothes, make them run as by magic, and get paid — not too bad, thanks for asking — at the end of the month?

Programming is tough. Where else do products from the most prestigious companies fail even in ordinary use? Where else does one find so many users complaining so loudly? Where else do engineers routinely work for hours or days trying to understand why something that should work doesn't?

Get ready for the mastery of programming and its professional form, software engineering; get ready for both the toughness and the fun.

SOFTWARE EVERYWHERE

By going into computing science you have chosen one of the most exciting and fast-moving topics in science and technology. Fifty years ago it was not even recognized as a scientific subject; today hardly a university in the world is without a CS department. Thousands of books, journals, magazines and conferences cover the field. The global revenues of its industry — called information technology or IT — are in the trillions. No other field, in the history of technology, has undergone growth of either such magnitude or such speed.

And we have made a difference. Without software there would be no large-scale plane travel, and in fact no modern planes (or modern cars, or high-speed trains) since their design requires sophisticated "Computer-Aided Design" software. To pay its workers, any large corporation would employ hundreds of people just to write the paychecks. A phone would still be a device tied to the wall by a cable. After taking a picture, you still could not see the result until the roll of film came back from processing. There would be no video games, no camcorders, no iPods and no iPhones, no Skype, no GPS to guide you to your destination even when there is no one around to ask. To produce a report you would still hand-write a draft, give it to a typist, and go through rounds of

correction requests. A sudden itch to know the name of the captain in *The Grand Illusion*, or the population of Cape Town, or the author of a familiar citation, would require (rather than typing a couple of search words and getting the answer in a blink) a trip to the library. The list goes on; at the heart of countless practices that now pervade our daily life lie programs — increasingly sophisticated programs.

All this does not happen by itself. While computers may have become a commodity, programs — without which computers would be useless — definitely are not. Programming, the task of constructing new programs or improving existing ones, is a challenging intellectual pursuit that requires programmers possessing creativity and experience. Through this book you will become familiar with the world of programs and programming, with a view to becoming a professional in the field.

CASUAL AND PROFESSIONAL SOFTWARE DEVELOPMENT

Although more and more people are acquiring basic computing proficiency, being able to program at a professional level is another matter, and is what a curriculum in computing science will bring you.

For comparison, consider mathematics. A few centuries ago, just being able to add and subtract 5-digit numbers required a university education, and in return provided qualifications for such good jobs as accountant. Nowadays these skills are taught in grade school; if you want to become an engineer or a physicist, or just a stock trader, you need to study more advanced mathematical topics, such as calculus, in a university. The boundary between basic training and university-level education has moved up.

Computing is following the same evolution, only much faster — the scale is in decades, not centuries. Not so long ago, being able somehow to program a computer was enough to land a job. Do not expect this today; an employer will not be much more impressed if your résumé states “I have written programs” than if you say you can add numbers.

What increasingly counts is the difference between having some basic programming experience and being a software engineer. The former skill will soon be available to anyone who has gone through a basic education; but the latter is a professional qualification, just like advanced mathematics. Studying this book is a step towards becoming a computing professional.

Factors that distinguish professional software development from casual programming include **size**, **duration** and **change**. In professional software development, you may become involved in programs that reach into the millions of lines of program text, must remain in operation for years or decades, and will undergo many changes and extensions in response to new circumstances. Many an issue that seems trivial or irrelevant when you are working on a medium-size program, meant only to solve a problem of immediate interest, becomes critical when you move to the scale of professional development.

With this book I'll try to prepare you for the real world of software, where systems are complex, solve serious problems (often affecting human life or property), stay around for a long time, and must lend themselves gracefully to requests for change.

PRIOR EXPERIENCE — OR NOT

This book does not assume any prior programming knowledge.

If you *did* program before, that experience will help you master the concepts faster. You will recognize some of the ideas, but you should also expect to be surprised at times, since the professional study of any topic is different from its use by the general public. Once in a while, for example, you may find that I belabor a seemingly simple matter. If so, you will (I think) discover after a while that the topic is not as simple as it seems at first, just as addition is more challenging to the mathematician than to the accountant. While you must be prepared to question some of your previous practices if they do not match the professional software engineering principles developed here, you can and should take advantage of everything you know. Learning to program well takes a lot of effort: every bit — every angle from which you can approach the problem — helps. In particular, the discussion relies, as explained below, on a supporting software system, Traffic. If you are familiar with programming and some programming languages, you will be able to discover some of Traffic by yourself, perhaps ahead of the official assignments. Do not hesitate to do so: one learns programming in part by reading existing programs for inspiration and imitation. You may have to do some guessing for elements of Traffic that rely on techniques and language constructs you have not formally studied yet, but this is where your experience will help you move faster.

On the other hand, if you have *not* done any programming, you're OK too. You might progress more slowly at the beginning, but should just study all the material carefully and do all the exercises. In particular, even though this book includes little actual mathematics, you will feel more comfortable if you have a mathematical mindset and the practice of logical reasoning. This is just as beneficial as programming experience, and will compensate for any handicap you feel relative to those fellow students in the back row who look like they typed their first program before they lost their baby teeth.

Programming, like the rest of computing science, is at the confluence of engineering and science. Success requires both a hands-on attitude (the "hacker" side, in the positive sense of the word), useful in technology-oriented work, and an ability to perform abstract, logical reasoning, required in mathematics and other sciences. Experience with programming helps you with the first goal; a logical mind helps you with the second. Wherever your strength lies, take advantage of it, and use this book to make up for any initial deficiency on the other side.

MODERN SOFTWARE TECHNOLOGY

Becoming a software professional requires more than one course or one book: it takes a multi-year curriculum in which — in addition to mathematical foundations such as logic and statistics — you will learn about software engineering, theory of computation, data structures, algorithms, operating systems, artificial intelligence, databases, hardware, networking, project management, software metrics, numerical computation, graphics and many other topics. But to prepare for these other computer science courses it is essential to use the best of what is known in software technology.

In recent years two major ideas, holding the potential for producing software of much better quality than was available before, have made their way into the software field: **object-oriented software construction** and **formal methods**. Both of these ideas, but especially the first, can be used to make the introductory study of computing more exciting and more profitable. Along with other concepts from modern software technology, they play a major role in this book. Let's have a quick advance look at both.

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

Object-oriented ("O-O") software construction follows from the realization that proper systems engineering must rely on a large inventory of high-quality reusable components, as in the electronic or construction industries. The O-O approach defines what form these components should have: each of them must be based on a certain *type of objects*. The term "object", which gives its name to the method, does not just refer to objects of the application domain, such as circles or polygons in a graphics program, but also to objects that are purely internal to the software, such as a list. If you do not quite see what this all means, that's normal; I hope that if you read this preface again in a few months it will all be crystal-clear!

A previous book ("Object-Oriented Software Construction", 2nd edition, Prentice Hall,1997) covers object technology in depth and at a more advanced level.

Object technology (the shorter name for object-oriented software construction) is quickly changing the software industry. Becoming familiar with it from the very beginning of your computing studies is an excellent insurance policy against technical obsolescence.

FORMAL METHODS

Formal methods are the application of systematic reasoning techniques, based on mathematical logic, to the construction of reliable software. Reliability, or rather the lack of it, is a vexing problem in software; errors, or the fear of error, are the programmer's constant companion. Anyone who uses computers has some anecdote about bugs.

Formal methods can help improve this situation. Learning formal methods in their full extent requires more knowledge than is available at the beginning of a university education. But the approach used in this book shows a significant influence of formal methods, in particular through the idea of *Design by Contract*, which considers the construction of software systems as the implementation of a number of individual contractual relations between modules, each characterized by a precise specification of obligations and benefits. I hope that you will understand the importance of these ideas and remember them for the rest of your career. In industry, everyone knows the difference between a programmer who just "hacks code" and one who is able to produce correct, robust and durable software elements.

LEARNING BY DOING

This book is not a theoretical presentation; it assumes that as you go along you practice what you learn on a computing system. The associated Web site provides links to the necessary software, in versions for Windows, Linux and other platforms, which you can download. Your school may also have the equivalent facilities available on its computers. In fact, the text prompts you, in some cases, to do the practical work with the software *before* learning the theoretical concepts.

touch.ethz.ch.

The system that you will use for this course is an advanced object-oriented environment: EiffelStudio, an implementation of the Eiffel analysis, design and programming language. Eiffel is a simple, modern language, used worldwide in large, mission-critical industrial projects (banking and finance, health care, networking, aerospace etc.) as well as for teaching and research in universities. The EiffelStudio version that you will use is exactly the same as the professional version, with the same graphical development environment and fundamental reusable components such as the EiffelBase, EiffelVision and EiffelMedia libraries. Your school may also have an academic license providing for maintenance and support.

Appendices present an introduction to four other languages widely used in industry: Java, C#, C++ and C. Any good software engineer must be fluent in several programming languages, including at least some of these; learning Eiffel will be a plus on your résumé (a mark of professionalism) and will help you master other object-oriented languages.

FROM THE CONSUMER TO THE PRODUCER

Because from day one of the course you will have the whole power of EiffelStudio at your fingertips, you will be able to skip many of the “baby” exercises that have traditionally been used to learn programming. The approach of this book is based on the observation that to learn a technique or a trade it is best to start by looking at the example of excellent work produced by professionals, and taking advantage of it by (in order) using that work, understanding its internal construction, extending it, improving it — and starting to build your own. This is the time-honored method of apprenticeship, which places newcomers under the guidance of experts.

The expertise is represented here by software, more specifically *library classes*: software elements from the Traffic library, specially developed for this book. As you write your first software examples, you will use these classes to produce results which are already impressive even though you have not had much to write; you will simply be relying on the mechanisms defined by the classes, acting, through your own software, as a *consumer* of existing components. Then, as someone who knows how to drive but is studying to become an automobile engineer, you will be encouraged to lift the hood and see how these classes are made, so that you can later on write extensions to the classes, improve them perhaps, and write your own classes.

The Traffic library, as its name suggests, provides mechanisms for dealing with traffic in a city — cars, pedestrians, metros, trams, taxis … — with graphical visualization, simulations, route computation, route animation etc. It is a rich reservoir of applications and extensions: you can build on it to write video games, solve optimization problems and try out many new algorithms.

The built-in examples use Paris as the sample city, because it is a popular tourist destination; you can easily adapt them to another city without touching the Traffic software, since all the location information is provided separately in a file (using a standard format, XML). It suffices to provide such a file representing your chosen city. For example, the course as taught at ETH Zurich uses the Zurich tram system, replacing the Paris metro.

ABSTRACTION

Basing your work on existing components has another important consequence for your education as a professional software engineer. The program modules that you reuse are a substantial piece of software, embodying a lot of knowledge. It would be very difficult to use them for your own applications if you had to read the full program text of each one you need. Instead, you will rely on a description of their *abstract interfaces*, which are extracted from their text (by automatic software mechanisms, part of EiffelStudio) but retain only the essential information that you need as a consumer. An abstract interface is a description of the purpose of a software module that only states its functions, not *how* the module's code realizes these functions. In software terminology it is also called the *specification* of the module, excluding the module's *implementation*.

This technique will help you learn one of the professional software developer's key skills: *abstraction*, meaning here the ability to distinguish the purpose of any piece of software from the details, often numerous, of its implementation. Every software development professor and textbook preaches the virtues of abstraction, and for good reason; here you will get the occasional bit of preaching too, but mostly you will be encouraged to learn abstraction by example, experiencing its benefits through the reuse of existing components. When you get to build your own software, you should apply the same principles; that is the only way to tame the ogre of software complexity.

The benefits of abstraction are quite concrete; you will experience them right from the beginning. The first program you will write is only a few lines long, but already produces a significant result (an animated itinerary on a city map). It can do this only by using modules from Traffic; and it can use them only because they are available through an abstract specification. If you had to examine the text of these modules (their *source code*), then the text of the modules they rely on themselves, directly or indirectly, you would quickly drown in an ocean of details and could not produce anything.

→"A class text", 2.1, page 15.

Throughout your work with software, abstraction is the lifevest that will save you from drowning in the sea of complexity.

DESTINATION: QUALITY

This book teaches not only techniques but methodology. Throughout the presentation you will encounter design principles and rules on programming style. Sometimes you may think that I am being fussy and that you could write the program just as well without the rules. Well, often you could. But the methodological rules make the difference between an amateurish program, which sometimes works, sometimes not, and the kind of production-quality software that you will want to produce. You should apply these rules not just because this book and your teachers say so, but because the power and speed of computers magnify any deficiency, however small, and require that the programmer pay attention to both the big picture and every detail. They are also good job insurance for your future career: there are many programmers around, and what really differentiates them in the eyes of an employer is the long-term quality of the software they produce.

Do not fool yourself with the excuse that "this is only an exercise" or "this is only a small program":

- Exercises are precisely where you need to learn the best possible techniques; when Airbus hires you to write the control software for their next plane, it will be too late.
- Calling a program "small" is often more hope than guarantee. In industry, many big programs are small programs that grew, since a good program tends to give its users endless ideas for requesting new functionalities.

So you should apply the same methodological principles to all the programs you develop, whether small or large, educational or operational.

Such is the goal of this book: not just to take you through the basics of software engineering and to let you experience the fun and thrill of producing software that works, but also to develop — along with a sense of beauty for the principles, methods, algorithms, data structures and other techniques that define the discipline — a sense for what makes good software stand out, and a determination to produce programs of the highest possible quality.

BM
Zurich / Santa Barbara, April 2009

*Instructor_preface**

**The preface for students is on page xiii.*

Right from its subtitle, this book shows its colors: it is not just about learning to program but about "Learning to Program *Well*". I am trying to get the students started on the right track so that they can enjoy programming — without enjoyment one does not go very far — and have a successful career; not just a first job, but a lifelong ability to tackle new challenges.

To help them reach this goal, the book applies innovative ideas detailed in the rest of this preface:

- **Inverted curriculum**, also known as the "outside-in" approach, relying on a large library of reusable components.
- Pervasive use of **object-oriented** and model-driven techniques.
- **Eiffel** and **Design by Contract**.
- A moderate dose of **formal methods**.
- Inclusion, from the very beginning, of **software engineering** concerns.

These techniques have for several years been applied to the "Introduction to Programming" course at ETH Zurich, taken by all entering Computer Science students. *Touch of Class* builds on this course and draws from its lessons. This also means that teachers using it as a textbook can rely on the teaching material developed for the course: slides, lecture schedules, exercises, self-study tutorials, student projects, even video recordings of our lectures.

← *See* "Community resources", page vii.

THE CHALLENGES OF A FIRST COURSE

This section is based on reference [12].

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 added themselves to the traditional ones:

- Adapting to ever higher stakes.
- Identifying the key knowledge and skills to teach.
- Coping with fads and outside pressures.
- Addressing a broad diversity of initial student backgrounds and abilities.
- Meeting high expectations for examples and exercises.
- Introducing the real challenges of professional software development.
- Teaching methodology and formal techniques without scaring off students.

The stakes are getting ever higher. When educating future software professionals, we must teach durable skills. It is not enough to present immediately applicable technology, for which in our globalized industry a cheaper programmer will always be available somewhere.

We must **identify the key knowledge and skills** to teach. Programming is no longer a rare, specialized ability; a large proportion of the population gets exposed to computers, software and some rudimentary form of programming, for example through spreadsheet macros or Web site development with Python, Ruby on Rails or ASP.NET. Software engineers need more than the ability to program; they must master software development as a professional endeavor, and by this distinguish themselves from the masses of occasional or amateur programmers.

It is important to keep a cool head in the presence of **fads and outside pressures**. Fads are a given of our field, and they are not always bad — structured programming, object technology and design patterns were all fads once — but we must make sure an idea has proved its mettle before inflicting it on our students. Outside pressures can be more delicate to handle. Student families have more say nowadays; this too is not necessarily bad, but sometimes results in inappropriate demands that we teach the specific technologies required in the job advertisements of the moment. What this attitude misses is that four years later some of the fashionable acronyms will be different, and that competent industry recruiters look for problem-solving skills, not narrow knowledge. It is our duty to serve the very interests of the students and their families by teaching them the fundamental matters, which will give them not just a first job but a rewarding career.

> This obsession with learning the right résumé-filling buzzwords for fear of not landing a job is silly anyway. It is a worldwide phenomenon, likely to last for decades, that a decent software developer has no trouble finding a good job. For all the gloom that the media have spread after the “burst of the Internet bubble”, and the fears that “all the jobs have gone to Bangalore”, no end is in sight to the challenges and excitement of our field, including of course for our colleagues in Bangalore. But there is a qualification: people who get and *keep* good jobs are not the narrow-minded specialists having been taught whatever filled the headlines of the day; they are the competent developers possessing a wide and deep understanding of computing science, and mastery of many complementary technologies.

The **broad diversity of student backgrounds** complicates the task. Among the students in the lecture hall on the first day of the introductory course, you will find some who have barely touched a computer, some who have already built an e-commerce site, and the full range in-between. What can the teacher do?

- It is tempting to assume a fair amount of prior programming experience and teach to the most advanced students only; but this shuts out students who simply have not had the opportunity or inclination to work with computers yet. In my experience, they include some who can later turn out to be excellent computer scientists thanks to excellent abstraction skills, which they have so far applied to topics such as mathematics rather than computing. The nerdy image still widely associated with computers may have prevented them from realizing that it is not about late-night video game sessions fueled by home-delivery pizza (a picture which, in particular, turns off many girls with excellent computer science potential) but about cogent thinking applied to solving some of the most exciting intellectual challenges open to humankind.
- We must not either — at the other extreme — bring everyone down to the lowest level: we need a way to catch and retain the attention of the more experienced students, letting them use and expand the insights they have already gained.

Reliance on reusable components, discussed below, is a central part of this book's solution to the issue. By giving students access to high-quality libraries, we let the novices take advantage of their functionality through abstract interfaces without needing at first to understand what's inside. The more advanced and curious students can, ahead of the others, start to peek into the internals of the components and use them as guidance for their own programs.

For this to work we need **high-quality examples**. Students today, having lived most of their lives in a world awash in the visual and auditory marvels of software-powered multimedia, expect to see and build more than small academic programs of the "Compute the 7-th Fibonacci number" kind. We must meet these expectations of the "Nintendo Generation" [3], without of course letting technological dazzle push aside the teaching of timeless skills.

> A variant of this issue is what we may call the "Google-and-paste" phenomenon, the name I use for what colleagues (generally using Java or C++ as their teaching language) report as follows: you give an exercise that calls for, say, a 100-line program solution. Internet-savvy students quickly find on the Web some Java code that does the job, except that it does much more as part of, maybe, a 10,000-line program. Now it does not take long for beginners to hit upon a key piece of programming wisdom from the ages: that if you see a program that works you mess with it as little as you can. You hold your breath when coming anywhere close to it. Following this insight, the student will just switch off (rather than remove) the parts he or she does not need, through a minimal set of changes. So the teacher gets a 10,000-line solution to an elementary question. Of course one may impose, if not a full prohibition of Web use (which in a computer science curriculum would be bizarre), precise rules that would exclude such a "solution". But how exactly? "Google-and-paste" is, after all, a form of reuse, even if not exactly the kind advocated by software engineering textbooks.

> The approach of this book goes one step further. Not only do we encourage reuse, we actually provide a large amount of code (150,000 lines of Eiffel at the time of writing) for reuse, and also for imitation since it is available in source form and explicitly designed as a model of good design and implementation. Reuse is one of the "best practices" promoted by the course from the beginning; but it is a form of reuse in line with principles of software engineering, based on abstract interfaces and contracts.

These questions contribute to the next issue on our list: **introducing the real challenges of professional software development**. In a university-level computer science or software engineering program, we cannot just teach programming in the small. We have to prepare students for what matters to professionals: programming in the large. Not all techniques that work well for small programs will scale up. The very nature of an academic environment, especially at an introductory level, makes it hard to introduce students to the actual challenges of today's industrial software: software developed by many people, expanding to many lines of code, adapted to many categories of uses and users, maintained over many years, and undergoing many changes.

This concern for scalability gives particular urgency to the last issue: **introducing methodology and formal reasoning without disconnecting from the students**. Methodological advice — injunctions to use information hiding, contracts and software engineering principles in general — can sound preachy and futile to beginners. Introducing some formal (mathematically-based) techniques, such as axiomatic semantics, can widen this potential gap between teacher and student. Paradoxically, the students who have already programmed a bit and stand to benefit most from such admonitions and techniques may be most tempted to discard them since they know from experience that it is possible — at least for small programs — to reach an acceptable result without strict rules. The best way to instill a methodological principle is pragmatic: by showing that it empowers you to do something that would otherwise be unthinkable, such as building impressive programs with graphics and animation. Our reliance on powerful libraries of reusable components is an example: right from the beginning of the course, students can produce significant applications, visual and all, thanks to these components; but they would never proceed beyond a few classes if as a prerequisite they had to read the code. The only reuse that works here is through abstract interfaces.

Rather than pontificating on abstraction, information hiding and contracts, it is better to let the students use these techniques and discover that they work. If an idea has saved you from drowning, you will not discard it as sterile theoretical advice.

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 does not in an introductory 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 fails to teach the system construction concepts that software engineers must master to be successful in professional development. Being able to produce programs is no longer sufficient; many non-professional software developers can do this honorably. What distinguishes the genuine professional is the mastery 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 order of this book is **outside-in**. It relies on the assumption that the most effective way to learn programming 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: a set of libraries, called Traffic, where the top layers have been produced specifically for this book, and the basic layers on which they rely (data structures, graphics, GUI, time and date, multimedia, animation…) are widely used in 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 API specifications, also known as *contract views*, which provide the essential information abstracted from the actual code. By relying on contract views, students are right from the start able 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". By the end of the course they should 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 in 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 topics 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 good introductory textbooks duly advocate abstraction, but the result of such exhortations is limited if all the student knows of programming is the usual collection of small algorithmic examples. I can lecture on abstraction too, but in the end, as noted earlier, the most effective way to convey the concepts is by example; by showing to the student how he or she can produce impressive applications through the reuse of existing software. That software is large at least by academic standards; trying to reuse it by reading the source code would take months of study. Yet students can, in the first week of the course, produce impressive results by reusing it through 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 right. It is the only way to survive when faced with an ambitious goal which you can only reach by standing on someone else's shoulders. Students who have gone early and often through this experience of building a powerful application through contract-based reuse of libraries do not need much more haranguing about abstraction and reuse; for them 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, and the resulting "Wow!".

The supporting software

Central to the Outside-In approach of this book is the accompanying Traffic software, available for free download. The choice of application area for the library required some care:

From touch.ethz.ch.

- The topic should be immediately familiar to all students, so that we can spend our time studying software issues and solutions, not the problem domain. (It might be fun to take, say, astronomy, but we would end up discussing comets and galaxies rather than inheritance structures and class invariants.)

- The area should provide a large stock of interesting algorithm and data structure examples, applications of fundamental computer science concepts, and new exercises that each instructor can devise beyond those in the book. This should extend beyond the introductory course, to enable our colleagues teaching algorithms, distributed systems, artificial intelligence and other computer science topics to take advantage of the software if they wish.
- The chosen theme should call for graphics and multimedia development as well as advanced graphical user interfaces.
- Unlike many video games, it must 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 is *transportation in a city*: modeling, planning, simulation, display, statistics. The supporting Traffic software is not just an application, doing a particular job, but a *library*, providing reusable components from which students and instructors can build applications. Although still modest, it has the basic elements of a Geographical Information System and the supporting graphical display mechanisms.

For its examples the book uses Paris, with its streets and transportation system; since the city's description comes from XML files, it is possible to retarget the example to any other city. (In the second week of the first session of the course at ETH a few students spontaneously provided a file representing the Zurich transportation network, which we have been using ever since.)

The very first application that the student produces takes up twelve lines. Its execution 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
            -- Show city info and route.
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

The algorithm includes only four instructions, and yet its effect is impressive thanks to the underlying Traffic mechanisms.

In spite of the reliance on an extensive body of existing software, I stay away from giving any impression of "magic". It is indeed possible to explain everything, at an appropriate level of abstraction. We should never say "*just do as you are told, you'll understand when you grow up*". This attitude is no better at educating students than it is at raising one's own children. In the first example as shown above, even the **inherit** clause can be explained in a simple fashion: I do not go into the theory of inheritance, of course, but simply tell the students that class *TOURISM* is a helper class introducing predefined 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 are also told that they do not need to look up the details of class *TOURISM*, but may do so if they feel the born engineer's urge to find out "how things work".

The rule, allowing our students to approach the topics progressively, is always to abstract and never to lie.

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, are developed over a long time, undergo many changes, and meet strong constraints of quality, timeliness and cost. Although the corresponding techniques are usually not taught to beginners, it is important to provide at least a first introduction, which appears in the last chapter. The topics include requirements analysis (the programmers we educate should not just be techies focused on the machinery but should also be able to talk to customers and understand their needs), facets of software quality, an introduction to lifecycle models, the concept of agile development, quality assurance techniques and Capability Maturity Models.

An earlier chapter complements this overview by presenting software engineering tools, including compilers, interpreters and configuration management systems.

Terminology

Lucid thinking includes lucid use of words. I have devoted particular attention to consistent and precisely defined terminology. The most important definitions appear in call-out boxes, others in the main body of the text.

At the end of each chapter a "New vocabulary" section lists all the terms introduced, and the first exercise asks the student to provide precise definitions of each. This is an opportunity to test one's understanding of the ideas introduced in the chapter.

TECHNOLOGY CHOICES

The book relies on a combination of technologies: an object-oriented approach, Design by Contract, Eiffel as the design and programming language. It is important to justify these choices and explain why some others, such as Java as the main programming language, were not retained.

Object technology

Many introductory courses now 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. Too often, for example, the first programs rely on static functions (in the C++ and Java sense of routines not needing a target object). There sometimes seems to be an implicit view that before being admitted to the inner chambers of modern technology students must suffer through the same set of steps that their teachers had to travel in their time. This approach retains the traditional bottom-up order, only reaching classes and objects as a reward to the students for having patiently climbed the *Gradus ad Parnassum* of classical programming constructs.

There is no good reason for being so fussy 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 is 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 played its part in fueling the writing of this book: *Life is uncertain — Eat dessert first*!

Classes and objects appear at the very outset and serve as the basis for the entire book. I have found that beginners adopt object technology enthusiastically if 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 in this presentation 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 as relativity yields 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 with traditional variables. So both the *static* architecture of programs and the *dynamic* structure of computations cover the traditional concepts. We absolutely want the students to master the traditional techniques such as algorithmic reasoning, variables and assignment, control structures, pointer manipulation (whose coverage here includes algorithms to reverse a linked list, a tricky task seldom covered in introductory courses), procedures and recursion; they must also be able to build entire programs from scratch.

Eiffel and Design by Contract

We rely on Eiffel and the EiffelStudio environment which students can download for free from www.eiffel.com. Universities can also install this free version (and purchase support if desired). This choice directly supports the pedagogical concepts of this book:

- The Eiffel language is uncompromisingly object-oriented.
- Eiffel provides a strong basis to learn other programming languages such as Java, C#, C++ and Smalltalk (as demonstrated by appendices which introduce the essentials of the first three of these languages, in about 30 pages each, by building on the concepts developed in the rest of the book).

→*Appendices* A *(Java),* B *(C#),* C *(C++).*

- Eiffel is easy for beginners to learn. 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, a debugger with unique reverse execution capabilities, automatic documentation (HTML or otherwise), software metrics, and leading-edge automatic testing mechanisms. It produces architectural diagrams automatically from the code; the other way around, it 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, Linux, Solaris and Microsoft .NET.
- EiffelStudio includes a set of carefully written libraries, which support the reuse concepts of this book, and serve as the basis of the Traffic library. The most relevant are: *EiffelBase*, which by implementing the fundamental structures of computer science supports the study of algorithms and data structures in part III: *EiffelTime* for date and time; *EiffelVision*, for portable graphics; and *EiffelMedia* for multimedia and animation.

- Unlike tools designed exclusively for education, Eiffel is used commercially for mission-critical applications handling tens of billions of dollars in investments, managing health care systems, performing civil and military simulations, and tackling other problems across a broad range of application areas. This is in my opinion essential to effective teaching of programming; a tool that is really good should be good for professionals as well as for novices.
- The Eiffel language is specified by a standard of the International Standards Organization. For the teacher relying on a programming language, an international standard, especially an ISO standard, is a guarantee of sustainability and precise definition.

For the text of the standard see tinyurl.com/y5abdx or the ECMA version (same contents, free access) at tinyurl.com/cq8gw.

- 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 round-trip Diagram Tool of EiffelStudio, is in line with the modeling benefits of object technology.

To support these goals, Eiffel directly implements the concepts of **Design by Contract**, which were developed together 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.

One should also not underestimate the role of syntax, for beginners as well as for experienced programmers. Eiffel's syntax — illustrated by the earlier short example — facilitates learning, enhances program readability, and fights mistakes:

← *Class PREVIEW, page* xxix.

- The language avoids cryptic symbols.
- Every reserved word is a simple English word, unabbreviated (*INTEGER*, not int).
- The equal sign =, rather than doing violence to hundreds of years of mathematical tradition, means the same as in math.
- Semicolons are not needed. In most of today's languages, program texts are peppered with semicolons terminating 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 obscure to beginners, they can be a source of errors. In Eiffel the semicolon as separator is optional, regardless of program layout. This leads to a neat program appearance, as you may see by picking any example in the book.

Encouraging such cleanliness in program texts should be part of the teacher's pedagogical goals. Eiffel includes precise style rules, explained along the way 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 language should let its users focus on the concepts rather than the notation. This is one of the goals of using Eiffel for teaching: that students should think about their problems, not about Eiffel

Why not Java?

Since courses in recent years have often used Java, or a Java variant such as C#, it is useful to explain why we do not follow this practice. Java is important for a computer scientist to know — indeed, as mentioned, the book provides an appendix describing Java, along with others on C#, C++ and C — but not suitable as a first teaching language. There is simply too much baggage to be learned before the student can start to think about the problems. This is apparent from the first program attempts; a Java "Hello World" reads

```
class First {
   public static void main(String args[])
   { System.out.println("Hello World!"); } }
```

This is full of irrelevant concepts, each an obstacle to learning. Why "public", "static", "void"? (Sure, I'll make my program *public* if you insist, but do you mean my efforts are *void* of any value?) These keywords have nothing to do with the purpose of the program, and the student won't begin to understand what they mean for a few months at least, yet he or she must include them, like magic incantations, for their programs to work. For the teacher this means repeatedly engaging in injunctions to use certain constructions without understanding what they mean. As noted earlier, this "*You'll understand when you grow up*" style is not good pedagogy. Eiffel protects us from it: we can explain every programming language construct that we use, right from the first example.

The object-oriented nature of Eiffel and the simplicity of the language play a role. It is ironic that every Java program, starting with the simplest example as shown above, uses a static function as its main program, departing from the object-oriented style of programming. There are of course people who do not like the idea of using O-O for the first course; but if you do choose objects, you should be consistent. At some point the students will realize that this fundamental scheme — the one you told them to use, from the first example to every subsequent one — is not object-oriented after all; how can you answer their inevitable question with a straight face?

Syntax, as noted, matters. In this first example the student must master strange symbol accumulations, like the final "'); } }", disconcerting to the eye and with no obvious role. In this accumulation the precise order of the symbols is essential, but is hard to explain and to remember. (Why a semicolon between a closing parenthesis and a brace? Is there a space after that semicolon, and if so how important is it?) Such aspects are troubling to beginners; inevitably, much time and effort are consumed learning them and recovering from trivial mistakes causing mysterious results, just when the student should be concentrating on the concepts of programming.

Another source of confusion is the use of "=" for assignment, inherited from Fortran through C and hard to justify in the twenty-first century. How many students starting with Java have wondered what value *a* must have for a = a + 1 to make sense, and, as noted by Wirth [15], why a = b does not mean the same as b = a ?

Inconsistencies are troubling: why, along with full words like "static", use abbreviations such as "args" and "println"? Students will retain from that first exposure to programming that it is not necessary to be consistent, and that saving keystrokes is more important than choosing clear names. (In the basic Eiffel library the operation to go to the next line is called *put_new_line*.) If indeed we later introduce methodological advice urging students to choose clear and consistent names, we can hardly expect them to take us seriously. "*Do as I say, not as I do*" is another dubious pedagogical technique.

To cite another example: when describing the need for a mechanism for treating operations as objects, like Eiffel's agents or the closures of other languages, I had to explain how one addresses the issue in a language such as Java that does not have these mechanisms. Since I used iterators as one of the motivating examples, I was at first happy to find that the original Sun page describing Java's "inner classes" also had code for an iterator design, which it would have been nice to use as a model. But then it includes declarations such as

→ *Chapter* 17.

See tinyurl.com/c4oprq (archive of java.sun.com/docs/ books/tutorial/java/ javaOO/ innerclasses.html, Oct. 2007; the page now uses a different example).

```
public StepThrough stepThrough() {
        return new StepThrough();
    }
```

I can perhaps try to justify this to seasoned programmers, but there is no way I can explain it to beginning students — and I admire anyone who can. Why does StepThrough appear three times? Does it denote the same thing each time? Is the change of letter case (StepThrough vs stepThrough) relevant? What does the whole thing mean anyway? Very quickly the introductory programming course

turns into painful exegesis of the programming language, with little time left for real concepts. In Alan Perlis's words, "*A programming language is low-level when its programs require attention to the irrelevant*".

Epigram #8, available at www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

Also contributing to the difficulties of using Java in an introductory course are the liberties that the language takes with object-oriented principles. For example:

- If *x* denotes an object and *a* one of the attributes of the corresponding class, you may by default write the assignment *x.a = v* to assign a new value to the *a* field of the object. This violates information hiding and other design principles. To rule it out, you must shadow every attribute with a "getter" function. For the teacher, the choice is between forcing students early on to add such noise to their programs, or let them acquire bad design habits which are then hard to unlearn.
- Java strictly distinguishes fully abstract modules, called *interfaces*, from fully implemented ones — classes. One of the benefits of the class mechanism, available as early as Simula 67, is to offer a full range of possibilities between these extremes. This idea is at the core of teaching the object-oriented method, in particular teaching design: you can express a notion, when you first identify it, as a fully *deferred* (abstract) class; then you refine it progressively, through inheritance, into a fully effective class. Classes at intermediate levels in this process are partially deferred and partially effective. Java does not let you use this approach if you may need to combine two or more abstractions through inheritance: all but at most one of the combined modules must be interfaces.

There are many more examples of such influences of Java on the teaching process; a new Eiffel user expressed a typical reaction by writing on a mailing list that "*I have written a lot of C++ and Java; all my brain power went on learning loads of nerdy computer stuff. With Eiffel I do not notice the programming and spend my time thinking about the problem.*"

A reason often invoked for using Java or C++ in introductory programming is the market demand for programmers in these languages. This is a valid concern, but it applies to the computer science curriculum as a whole, not to the first course. Programming at the level required of a CS graduate today is hard enough; we should use the best pedagogical tools. If market demand had been the determinant, we would never in the past have used Pascal (for many years the introductory language of choice), even less Scheme. Following the trends reflected in the latest ads for programmers we would in turn have imposed Fortran, Cobol, PL/I, Visual Basic, maybe C — and trained programmers who, a few years after graduation, would have found their skills obsolete when the great wheel of fashion turned. Our duty is to train problem-solvers who can quickly adapt to the evolutions of our discipline.

We should not let short-term market considerations damage pedagogical principles. In other words: if you think Java or C++ are ideal teaching tools, use them; you probably will not like this book very much anyway. But if you agree with its approach, do not let yourself be scared that some student or parent will complain that you use an "academic" approach. Explain to them that you are teaching programming in the best way you know, that someone who understands programming will retain that skill for life, and that any half-decent software engineer can pick up a new programming language at breakfast — in case he or she has not already picked it up from other courses of your curriculum. As to the "academic" qualification (assuming that in a university context, it is meant as derogatory!), point them to eiffel.com and its long list of mission-critical systems in Eiffel in the financial industry, aerospace, defense, networking, computer-aided design, health care and other areas, successfully deployed by major companies, often after attempts in other languages had failed.

Java, C#, C++ and C are, for the next few years, an important part of any software engineer's baggage; it is important, as reflected by this book's four language-specific appendices, to ensure that the students know them. This goal is, however, unrelated to the question of what techniques to use in the introductory course. Students will most likely be exposed to these languages at some point; it would be a rare curriculum these days where no course uses at least one of them. In any case, no introductory course that I know covers *all* of them, so students need to learn more regardless of the initial teaching language.

In our surveys [13], about 50% of students have used Java or C++ before *they reach the introductory course.*

Programming languages and the programming culture associated with each of them are interesting objects of study. Our group at ETH, which teaches introductory programming in Eiffel, has introduced courses for the third year and beyond, devoted to specific languages: "Java in Depth", "C# in Depth" etc.

Once you understand the concepts of programming, you are well prepared to master diverse languages. Eiffel is a benefit here too: as many people have noted, having learned Eiffel and its object model helps you become a better C++ or Java programmer.

> As a potential employer in both academia and industry I see dozens of CVs every month. They all boast of the same skills, including C++ and Java. Other than as checkboxes to be ticked, this will not impress anyone. What recruiters do watch for is any skill that sets out an applicant from the hordes of others with similar backgrounds. An example of such a distinctive advantage is that the applicant knows a fully object-oriented approach with support for software engineering, as evidenced by a curriculum using Eiffel and Design by Contract. It is possible to survive a C++-based curriculum without ever understanding O-O concepts in any depth; with Eiffel that is less likely. Competent employers know that what counts, beyond immediate skills, is depth of understanding of software issues and aptitude for long-term professional development. All the effort deployed through this book and its use of Eiffel is directed at these goals.

It may be appropriate here to cite Alan Perlis again: *A language that doesn't affect the way you think about programming is not worth knowing.* *Epigram #19.*

HOW FORMAL?

One of the benefits of the Design by Contract approach is to expose the students to a gentle dose of "formal" (mathematics-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, based on a mathematical specification language. In addition — although not as a substitute for such a course — the ideas should influence the entire software curriculum, even though as discussed below it is not desirable today to subject beginners to a fully formal approach. The challenge is not only to include an introduction to formal reasoning along with 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 software architecture, 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 open the way for the introduction of formal methods, which they will fully appreciate when they have acquired more experience with programming.

→ *In chapter 4.*

In no way does the use of a mathematical basis imply a stiff or intimidating manner. Some formality in the concepts goes well with a practical, hands-on approach. For example the text introduces loops as an *approximation* mechanism, to compute a solution on successively larger subsets of the data; in this view the notion of *loop invariant* comes naturally, at the very beginning of the discussion of loops, as a key property stating the approximation obtained at every stage.

This emphasis on practicality distinguishes Design by Contract from the fully formal approaches used in some introductory courses, whose teachers hold that students should first learn programming as a mathematical discipline. Sometimes they go so far as to keep them away from the computer for a semester or a full year. The risk of such dogmatism is that it may produce the reverse of its intended effect.

Students, in particular those who have programmed before, realize that they can produce a program — not a perfect program, but a program — without a heavy mathematical apparatus; if you tell them that it's not possible they will just disconnect: they may from then on reject any formal technique as irrelevant, including both simple ideas which can help them now and more advanced ones later. As Leslie Lamport — not someone to be suspected of underestimating the value of formal methods — points out [6]:

> [*In American universities*] *there is a complete separation between mathematics and engineering. I know of one highly regarded American university in which students in their first programming course must prove the correctness of every tiny program they write. In their second programming course, mathematics is completely forgotten and they just learn how to write C programs. There is no attempt to apply what they learned in the first course to the writing of real programs*.

Our experience confirms this. First-year students, who react well to Design by Contract, are not ready for a fully formal approach. To develop a real appreciation for its benefits you must have encountered the difficulties of industrial software development. On the other hand, it also does not work to let students develop a totally informal approach first and, years later, suddenly reveal that there is more to programming than hacking. The appropriate technique, I believe, is incremental: introduce Design by Contract techniques right from the start, with the associated idea that programming is based on a mathematical style of reasoning, but without overwhelming students with concepts beyond their reach; let them master the practice of software development on the basis of this moderately formal approach; later in the curriculum, bring in courses on such topics as formal development and programming language semantics. This cycle can be repeated, as theory and practice reinforce each other.

Such an approach helps turn out students for whom correctness concerns are not an academic chimera but a natural, ever-present component of the software construction process.

In the same spirit, the discussion of high-level functional objects (agents, chapter 17, and their application to event-driven programming in chapter 18) provides the opportunity of a simple introduction to **lambda calculus**, including currying — mathematical topics that are seldom broached in introductory courses but have applications throughout the study of programming.

OTHER APPROACHES

Looking around at university curricula, talking to teachers and examining textbooks leads to the observation that four main approaches exist today for introductory programming:

1 Language-focused.

2 Functional (in the sense of functional programming).

3 Formal.

4 Structured, Pascal or Ada-style.

It is important to understand the benefits of these various styles — indeed we retain something from each of them — and their limitations.

The first approach is probably the most common nowadays. It focuses on a particular programming language, often Java or C++. This has the advantage of practicality, and of easily produced exercises (subject to the Google-and-Paste risk), but gives too much weight to the study of the chosen language at the expense of fundamental conceptual skills. Relying on Eiffel helps us teach the concepts, not the specifics of a language.

The second approach is illustrated in particular by the famous MIT course based on the Scheme functional programming language [1], which has set the standard for ambitious curricula; there also have been attempts using Haskell, ML or OCaml. This method is strong on teaching the logical reasoning skills essential to a programmer. We strive to retain these benefits, as well as the relationship to mathematics, present here through logic and Design by Contract. But in my opinion object technology provides students with a better grasp of the issues of program 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.

While, as noted, the curriculum should not be a slave to the dominant technologies just because they are dominant, using techniques too far removed from practice subjects us to the previously mentioned risk of disconnecting from the students, especially the most advanced ones, if they see no connection between what they are being taught and what their incipient knowledge of the discipline tells them. (Alan Perlis put this less diplomatically: *Purely applicative languages are poorly applicable*.)

Epigram #108.

I would argue further 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 the most difficult issues disappear. If this view is correct, we are not particularly helping students by protecting them from these aspects at the beginning of their education, presumably abandoning them to their own resources when they encounter them later. (Put in a different way: functional programming seems to require monads these days and, given a choice, I'd rather teach assignment than category theory.)

It is useful to point out that O-O programming is as mathematically respectable — through the theory of abstract data types on which it rests and, in Eiffel, the reliance on contracts — and as full of intellectual challenges as any other approach. Recursion, one of the most fascinating tools of functional programming, receives extensive coverage in the present book.

→ *Chapter* 14.

Some of the comments on functional programming also apply to the third approach, reliance on formal methods. As discussed above, a fully formal approach is, at the introductory programming level, premature. The practical effect may be to convince students that academic computer science has nothing to do with the practice of software engineering, and lead them to a jaded, method-less approach to programming.

The fourth commonly used approach, pioneered at ETH, draws its roots in the structured programming work of the seventies, and is still widespread. It emphasizes program structure and systematic development, often top-down. The supporting programming language is typically Pascal, or one of its successors such as Modula-2, Oberon or Ada. The approach of this book is heir to that tradition, with object technology viewed as a natural extension of structured programming, and a focus on programming-in-the-large to meet the challenges of programming in the new century.

TOPICS COVERED

The book is divided into five parts.

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 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

→ *Chapter* 5.

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 and the tricky issues of working with linked structures, only after program structuring concepts.

Part II, entitled “How things work”, presents the internal perspective. It starts with the basics of computer organization (covered from the viewpoint of a programmer and including essential concepts only), syntax description methods (BNF and its applications), programming languages and programming tools. The two chapters that follow cover core topics: syntax and how to describe it, including BNF and an introduction to the theory of finite automata; and an overview of programming languages, programming tools and software development environments.

Part III examines fundamental data structure and algorithm techniques. It is made of three chapters:

- Fundamental data structures — not a substitute for the “Data Structures and Algorithms” course which often follows the introductory course, but introducing genericity, algorithm complexity, and several important data structures such as arrays, lists of various kinds and hash tables.
- Recursion, including binary trees (in particular binary search trees), an introduction to fixpoint interpretations, and a presentation of techniques for implementing recursion.
- A detailed exploration of one interesting algorithm family, topological sort, chosen for its many instructive properties affecting both algorithm design and software engineering. The discussion covers the mathematical background, the progressive development of the algorithm for efficient execution, and the engineering of the API for convenient practical use.

Part IV goes into the depth of object-oriented techniques. Its first chapter covers inheritance, addressing many details seldom addressed in introductory courses, such as the Visitor pattern (which complements basic inheritance mechanisms for the case of adding operations to existing types). The next chapter addresses a technique that is increasingly accepted as a required part of modern object-oriented frameworks: function objects, also known as closures, delegates and *agents* (the term used here). It includes an introduction to lambda calculus. The final chapter in this part applies agent techniques to an important style of programming: event-driven computation. This is the opportunity to review another design pattern, Observer, and analyze its limitations.

Part V adds the final dimension, beyond mere programming, by introducing concepts of software engineering for large, long-term projects.

Appendices, already mentioned, provide an introduction to programming languages with which students should be familiar: Java, C#, C++ — a bridge between the C and O-O worlds — and C itself.

ACKNOWLEDGMENTS

A number of elements of this Instructor's Preface are taken from earlier publications: [7], [8], [9], [10], [12].

This book has its source, as noted, in the "Introduction to Programming" course at ETH Zurich and would not have been possible without the outstanding environment provided by ETH. Both the course and the book exist as a result of Olaf Kübler's trust (or wager) that in addition to entrepreneur I could also be a professor. Specific thanks go to the Rectorate (which financed the initial development of the Traffic library), to the Rector himself, Konrad Osterwalder, and to the computer science department, particularly Peter Widmayer who, as then department head, first asked me whether I would like to teach introductory programming, and made the effort of coordinating his own course with mine.

I have taught the course every Fall since 2003 and am indebted to the outstanding assistant team that has built an effective operation for handling exercise sessions, supporting students, devising exercises and exams, grading them, organizing student projects, writing supplementary documents and teaching aids, and on the odd occasion substituting for me in lectures. This has enabled me to concentrate on developing the pedagogical concepts and the core material, reassured that the logistics would work. I am also grateful to the hundreds of students who have taken this course, put up with my trials and errors, and provided feedback, including the best kind of feedback one can hope for: excellent software projects.

See e.g. games.ethz.ch.

The course assistants, 2003-2008, have been: Volkan Arslan, Stephanie Balzer, Till Bay, Karine Bezault (Karine Arnout), Benno Baumgartner, Rolf Bruderer, Ursina Caluori, Robert Carnecky, Susanne Cech Previtali, Stephan Classen, Jörg Derungs, Ilinca Ciupa, Ivo Colombo, Adam Darvás, Peter Farkas, Michael Gomez, Sebastian Gruber, Beat Herlig, Matthias Konrad, Philipp Krähenbühl, Hermann Lehner, Andreas Leitner, Raphael Mack, Benjamin Morandi, Yann Müller, Marie-Hélène Nienaltowski (Marie-Hélène Ng Cheong Vee), Piotr Nienaltowski, Michela Pedroni, Marco Piccioni, Conrado Plano, Nadia Polikarpova, Matthias Sala, Bernd Schoeller, Wolfgang Schwedler, Gabor Szabo, Sébastien Vaucouleur, Yi (Jason) Wei and Tobias Widmer. While I should cite virtually all members of the ETH Chair of Software Engineering for their support and ideas I must at least single out Manuel Oriol for his participation in our education research, Till Bay (for his development of the EiffelMedia library, the basis for so many student projects, of the EiffelVision drawables of Traffic in his diploma thesis, and of the Origo project hosting site at origo.ethz.ch as part of his PhD thesis), Karine Bezault, Ilinca Ciupa, Andreas Leitner, Michela Pedroni and Marco Piccioni (all of them head assistants at some point and helpful in many other ways). Claudia Günthart provided excellent administrative support.

The Traffic software has a particularly important role in the approach of this book. The current version was developed over several years by Michela Pedroni, starting from an original version written by Patrick Schönbach under the management of Susanne Cech Previtali; a number of students contributed to the software, supervised by Michela in various semester and master's projects, in particular (in approximate chronological order) Marcel Kessler, Rolf Bruderer, Sibylle Aregger, Valentin Wüstholz, Stefan Daniel, Ursina Caluori, Roger Küng, Fabian Wüest, Florian Geldmacher, Susanne Kasper, Lars Krapf, Hans-Hermann Jonas, Michael Käser, Nicola Bizirianis, Adrian Helfenstein, Sarah Hauser, Michele Croci, Alan Fehr, Franziska Fritschi, Roger Imbach, Matthias Loeu, Florian Hotz, Matthias Bühlmann, Etienne Reichenbach and Maria Husmann. Their role was essential in bringing the user perspective to the product, as most of them had previously taken the introductory course with early versions of Traffic. Michela Pedroni was also instrumental in reconciling the software with the book and the other way around and, more generally, in helping develop the underlying pedagogical approach — inverted curriculum, outside-in, tool support (see trucstudio.origo.ethz.ch). Marie-Hélène Nienaltowski also participated in our pedagogical work, provided the TOOTOR system to help students master the material, and tried out the approach at Birkbeck College, University of London.

I am grateful to my colleagues in the Computer Science Department (Departement Informatik) at ETH for many spirited discussions about the teaching of programming; I should acknowledge in particular the criticism and suggestions of Walter Gander (who also helped me improve an important numerical example), Gustavo Alonso, Ueli Maurer, Jürg Gutknecht, Thomas Gross, Peter Müller and Peter Widmayer. Beyond ETH, I benefited from many discussions with educators including Christine Mingins, Jonathan Ostroff, John Potter, Richard E. Pattis, Jean-Marc Jézéquel, Vladimir Billig, Anatoly Shalyto, Andrey Terekhov and Judith Bishop.

Like all my work of recent years, this book has a huge debt to the outstanding work of developing the EiffelStudio environment and libraries at Eiffel Software under the leadership of Emmanuel Stapf and with the participation of the entire development team. I am also grateful to the willingness of the ECMA International TC49-TG4 standard committee, in charge of the ISO Eiffel standard, to take into consideration the needs of beginning students when discussing improvements and extensions to the language design; the debt here is to Emmanuel Stapf again, Mark Howard, Éric Bezault, Kim Waldén, Zoran Simic, Paul-Georges Crismer, Roger Osmond, Paul Cohen, Christine Mingins and Dominique Colnet. Discussions on the Eiffel Software user list have also been most enlightening.

groups.eiffel.com.

Listing even a subset of the people whose work has influenced the present one would take many pages. Many are cited in the text itself but one is not: the presentation of recursion owes some of its ideas to the online record of Andries van Dam's lectures at Brown.

Many people provided comments on drafts of the book; I should in particular note Bernie Cohen (although his principal influence on this book occurred many years earlier, when he proposed the concept of inverted curriculum), Philippe Cordel, Éric Bezault, Ognian Pishev and Mohamed Abd-El-Razik, as well as ETH students and assistants Karine Bezault, Jörg Derungs, Werner Dietl, Moritz Dietsche, Luchin Doblies, Marc Egg, Oliver Jeger, Ernst Leisi, Hannes Röst, Raphael Schweizer and Elias Yousefi. Hermann Lehner contributed several exercises. Trygve Reenskaug contributed important and perceptive comments on the event-driven design chapter. I am particularly grateful for the extensive reading and error reporting that Marco Piccioni and Stephan van Staden performed on chapters of the last drafts.

Special thanks are due to the originators of the material from which the language-specific appendices is drawn: Marco Piccioni (Java, appendix A), Benjamin Morandi (C#, appendix B) and Nadia Polikarpova (C++, appendix C). I obviously remain responsible for any deficiency in the resulting presentations.

I cannot find strong enough words to describe the value of the extremely diligent proofreading of the final version by Annie Meyer and Raphaël Meyer, resulting in hundreds (actually thousands) of corrections and improvements.

Since so many people have helped I am afraid I am forgetting some, and will keep a version of this section online, correcting any omissions. I do want to end, however, by acknowledging the help and advice of Monika Riepl, from le-tex publishing services in Leipzig, on typesetting issues, and the warm and efficient support, throughout the publishing process, of Hermann Engesser and Dorothea Glaunsinger from Springer Verlag.

See touch.ethz.ch/acknowledgments.

BM
Santa Barbara / Zurich, April 2009

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] Joint Task Force on Computing Curricula: *Computing curricula 2001* (final report). December 2001, tinyurl.com/d4uand.

[5] Joint Task Force for Computing Curricula 2005: *Computing Curricula 2005*, 30 September 2005, www.acm.org/education/curric_vols/CC2005-March06Final.pdf.

[6] Leslie Lamport: *The Future of Computing*: *Logic or Biology*; text of a talk given at Christian Albrechts University, Kiel on 11 July 2003, research.microsoft.com/users/lamport/pubs/future-of-computing.pdf.

[7] 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.

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

[9] Bertrand Meyer: *Software Engineering in the Academy*, in Computer (IEEE), vol. 34, no. 5, May 2001, pages 28-35,se.ethz.ch/~meyer/publications/computer/academy.pdf.

[10] Bertrand 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.

[11] 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 and 74.

[12] Michela Pedroni and Bertrand Meyer: *The Inverted Curriculum in Practice*, in *Proceedings of SIGCSE 2006* (Houston, 1-5 March 2006), ACM, se.ethz.ch/~meyer/publications/teaching/sigcse2006.pdf.

[13] Michela Pedroni, Manuel Oriol and Bertrand Meyer: *What do Beginning CS students know?*, submitted for publication, 2009.

[14] Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*, in *Proceedings of Tenth Australasian ComputingEducation Conference (ACE2008)*, Wollongong, January 2008), crpit.com/confpapers/CRPITV78Lister.pdf.

[15] Niklaus Wirth: *Computer Science Education*: *The Road Not Taken*, opening address at ITiCSE conference, Aarhus, Denmark, June 2002, www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm.

Web addresses come and go. All URLs appearing in this bibliography and the rest of the book were operational on April 19, 2009.

Note to instructors: what to cover?

To provide flexibility for the instructor, the book has more material than will typically be covered in a one-semester course. The following is my view of what constitutes essential material and what can be viewed as optional. It is based on my experience and will naturally need to be adapted to every course's specifics and every instructor's taste.

- Chapters 1 to 4 should probably be covered in their entirety, as they introduce fundamental concepts.
- Chapter 5 on logic introduces fundamental concepts. If students are also taking a logic course the material can be covered briefly, with a focus on relating computer scientists' and logicians' notations and conventions. I find it useful to insist on the properties of implication, initially counter-intuitive to many students ("Getting a practical feeling for implication", page 86); also, the course should discuss **semistrict boolean operators** (5.3), which logicians usually do not cover.
- Chapter 6 on object creation is necessary for the rest of the presentation.
- So is chapter 7 on control structures up to 7.6; the remaining sections present details of the low-level branching structure and some language variants. You should mention structured programming (7.8).
- Chapter 8 on routines should in my view be included in its entirety; in particular it is useful to provide a simple proof of the undecidability of the Halting Problem.
- In chapter 9, sections up to 9.5 cover fundamental concepts. 9.6, discussing the difficulty of programming with references, with the example of list reversal, is important but more advanced. The last subsection, on dynamic aliasing, is optional material.
- How much to cover chapter 10 on computers depends on what students are learning elsewhere about computer architecture. The chapter is not deep but provides basic points of reference for programmers.
- Chapter 11 on syntax is important material but not absolutely required for the rest of the book. I suggest covering at least the sections up to 11.4 (if only because students need to understand the concept of abstract syntax). If most students will *not* take a course on language and compilers, they will benefit from the basic concepts in subsequent sections.
- Chapters 12 on programming languages and tools is background material; I do not cover it explicitly in my class but provide it as a resource.
- Chapter 13 introduces fundamental concepts on data structures, genericity, static typing and algorithm complexity. It is possible to skip 13.8 (list variants) and 13.13 (iteration).

- Chapter 14 discusses recursion in some depth — more depth than is customary in an introductory presentation, because I feel it is useful to remove the potential mystery of recursive algorithms and show the importance of recursion *beyond* algorithms: recursive definitions, recursive data structures, recursive syntax productions and recursive proofs. The core material is the beginning of the chapter: 14.1 to 14.4, including the discussion of binary trees. The other sections may be viewed as supplementary; backtracking and alpha-beta (14.5) are a useful illustration of the applications of recursion. If the course is strongly implementation-oriented, consider 14.9 (implementing recursion); if you think that contracts are important, direct the students to 14.8 (contracts and recursion).
- Chapter 15 is a detailed discussion of an important application, topological sort. It introduces no new programming construct and so you can skip it, or replace it with one of your own examples, without damage. I cover it in some depth because it describes the complete progression from mathematics to algorithms to choice of optimal data structures to proper engineering of the API.
- In chapter 16, on inheritance, the essential sections are 16.1 to 16.7, plus 16.9 on the role of contracts, which illuminates the whole concept of inheritance. It is also useful to explain the connection to genericity in 16.12. The end of the chapter, in particular 16.14 about the Visitor pattern, is more advanced material that most courses probably will not have the time to cover, but which can be given as a reading assignment or as preparation for later courses.
- Chapter 17 on agents (closures, delegates) again goes beyond the usual scope of introductory courses. This is so important to modern programming that in my opinion it should be covered at least up to 17.4 (including illustrations through numerical programming and iteration). I usually do not have the time to cover 17.6, a gentle introduction to lambda calculus, but it should interest the more mathematically-oriented students, if only as extra reading material.
- If you do cover agents, you should then reap the benefits by covering the application to event-driven programming and especially GUI design (of interest to many students) in chapter 18. This is a good opportunity to learn an important pattern, Observer. Our course covers this and the previous chapter together, in four 45-minute lectures.
- Chapter 19 (introduction to software engineering) is not critical to an introductory course and I have not had time so far to cover it (but we do have "software architecture" and "software engineering" courses later in the curriculum). It is appropriate for an audience that needs to be exposed to the issues of production-quality software development in industry.
- The appendices are background material and I do not cover them, although some instructors might want to devote some time to a language such as Java or C++ (we do this, as noted, in specialized courses focusing on these languages).

A final note: while the course and the book were developed together, I always make a point of devoting a couple of lectures in the course to a topic *not* covered in the book — to introduce some spontaneity and avoid limiting the course to pre-packaged material. I like for example to present the algorithm for *Levenshtein distance* (edit distance between two strings), as it provides an outstanding example of the usefulness of loop invariants: without the invariant the algorithm looks like magic, with the introduction of the invariant it becomes limpid. Some of the extra material is available from the book site, touch.ethz.ch. (In the same vein, I have found that the textbook is sufficiently detailed to allow me to use a "Socratic" style for a couple of lectures in the semester: I ask the students to read a chapter in advance; then I do not cover the material sequentially in class but just come and wait for questions. Maybe this can work for other instructors as well.)