Paradigmi di programmazione

nella didattica dell'informatica

Selezione di riferimenti ed estratti raccolti da Claudio Mirolo

I passi che seguono sono ripresi dalle prefazioni e dalle guide per gli insegnanti (in qualche caso l'indice dei contenuti) e si riferiscono a corsi introduttivi all'informatica di livello accademico. I materiali sono stati scelti come esempi rappresentativi della varietà di approcci, pur senza pretesa di esaustività, in quanto si caratterizzano per una chiara impostazione di fondo, per l'originalità o per l'approccio innovativo. Benché non siano indirizzati direttamente agli allievi della scuola superiore, la conoscenza delle proposte didattiche ivi contenute e della filosofia che li ispira interessa anche l'insegnante che vuole programmare corsi di informatica e può fornire una valida guida ai fini dell'insegnamento dell'informatica a qualsiasi livello scolastico.

Laddove possibile, ho anche tentato di classificare i riferimenti in relazione ai 6 principali approcci metodologici identificati nel documento "ACM/IEEE-CS Computing Curricula 2001" da parte delle principali istituzioni che si occupano dei curricula di informatica a livello internazionale, ma si può notare la diversità di impostazione anche all'interno di uno stesso ambito. In alcuni casi la collocazione è dichiarata esplicitamente dagli autori, o comunque evidente. In altri gli obiettivi non sono univoci, coinvolgendo elementi caratterisitci dell'uno o dell'altro approccio. In questi casi ho soggettivamente scelto di privilegiare un particolare punto di vista rispetto agli altri possibili. Consultando i materiali, comunque, sarà facile capire la maggiore articolazione dei contributi.

Fatta forse eccezione per l'approccio *breadth-first*, che si presta a diverse interpretazioni, in tutti gli altri casi il ruolo della programmazione è importante fin dall'inizio. Nei casi *imperative-first*, *functional-first* e *object-first* esso costituisce la chiave della didattica dell'informatica: l'impostazione è caratterizzata esplicitamente dalla scelta di uno specifico paradigma e di un linguaggio della corrispondente classe. L'approccio *algorithms-first* tende a svilupparsi a un livello più astratto, quello algoritmico, che chiaramente presuppone un modello computazionale non dissimile da quelli soggiacenti ai linguaggi di programmazione, quasi sempre privilegiando il paradigma imperativo, cosicché la principale distinzione sta piuttosto nel livello di dettaglio con cui i programmi vengono formalizzati. Infine, l'approccio *hardware-first* si caratterizza per il fatto che affronta un modello computazionale (imperativo) a partire dai suoi mattoni elementari, e quindi introduce dapprima un linguaggio di programmazione a basso livello, per poi passare a un linguaggio come C.

Assumendo l'ottica dei possibili paradigmi di programmazione, la classificazione a cui ho accennato non considera la programmazione logica, che pure a cavallo fra gli anni ottanta e novanta era risuscita a sollevare entusiasmi ed era stata oggetto di attenzione anche in ambito scolastico. Attualmente sembra finita al margine, almeno come strumento per introdurre efficacemente l'informatica, e perciò non si parla di un approccio *declarative-first*.

Ho preferito raccogliere alcuni estratti, piuttosto che fornire semplicemente i riferimenti *sitografici*, pensando che possa essere utile avere sott'occhio degli *abstract* in un unico documento, per facilitare il lavoro almeno in una prima fase di orientamento. Per gli eventuali approfondimenti successivi, invece, sarà opportuno seguire i rimandi.

Introduzione all'informatica: approccio imperativo / 1

Jeri R. Hanly, Elliot B. Koffman, "Problem Solving and Program Design in C", 2007.

`http://www.aw-bc.com/catalog/academic/product/0,1144,0321409914,00.html`

This textbook teaches a disciplined approach to solving problems and to applying widely accepted software engineering methods to design program solutions as cohesive, readable, reusable modules. We present as an implementation vehicle for these modules a subset of ANSI C [...]. In preparing this edition, we have added [a chapter] which can serve as a transition to the study of C++ in a subsequent course. [...] Two of our goals—teaching program design and teaching C—may be seen by some as contradictory. C is widely perceived as a language to be tackled only after one has learned the fundamentals of programming in some other, friendlier language. The perception that C is excessively difficult is traceable to the history of the language. Designed as a vehicle for programming the UNIX operating system, C found its original clientele among programmers who understood the complexities of the operating system and the underlying machine, and who considered it natural to exploit this knowledge in their programs. Therefore, it is not surprising that many textbooks whose primary goal is to teach C expose the student to program examples requiring an understanding of machine concepts that are not in the syllabus of a standard introductory programming course. In this text we are able to teach both a rational approach to program development and an introduction to ANSI C because we have chosen the first goal as our primary one. One might fear that this choice would lead to a watered-down treatment of ANSI C. [...]

The order in which C language topics are presented is dictated by our view of the needs of the beginning programmer rather than by the structure of the C programming language. The reader may be surprised to discover drat there is no chapter entitled "Pointers." This missing chapter title follows from our treatment of C as a high-level language, not from a lack of awareness of the critical role of pointers in C. Whereas other high-level languages have separate language constructs for output parameters and arrays, C openly folds these concepts into its notion of a pointer, drastically increasing the complexity of learning the language. We simplify the learning process by discussing pointers from these separate perspectives where such topics normally arise when teaching other programming languages, thus allowing a student to absorb the intricacies of pointer usage a little at a time. Our approach makes possible the presentation of fundamental concepts using traditional high-level language terminology—output parameter, array, array subscript, string—and makes it easier for students without prior assembly language background to master the many facets of pointer usage.

[...] The book presents many aspects of software engineering. Some are explicitly discussed and others are taught only by example. [...]

Table of Contents

Introduzione all'informatica: approccio imperativo / 2

Walter Savitch, "Problem Solving with C++", 2007.

`http://www.aw-bc.com/catalog/academic/product/0,1144,0321412699,00.html`

Table of Contents

Introduzione all'informatica: approccio funzionale / 1

Max Hailperin, Barbara Kaiser, Karl Knight, "Concrete Abstractions", 1999.

`http://www.gustavus.edu/+max/concrete-abstractions.html`

At first glance, the title of this book is an oxymoron. After all, the term abstraction refers to an idea or general description, divorced from physical objects. On the other hand, something is concrete when it is a particular object, perhaps something that you can manipulate with your hands and look at with your eyes. Yet you often deal with concrete abstractions. Consider, for example, a word processor. When you use a word processor, you probably think that you have really entered a document into the computer and that the computer is a machine which physically manipulates the words in the document. But in actuality, when you "enter" the document, there is nothing new inside the computer—there are just different patterns of activity of electrical charges bouncing back and forth. Moreover, when the word processor "manipulates" the words in the document, those manipulations are really just more patterns of electrical activity. Even the program that you call a "word processor" is an abstraction—it's the way we humans choose to talk about what is, in reality, yet more electrical charges. Still, although these abstractions such as "word processors" and "documents" are merely convenient ways of describing patterns of electrical activity, they are also things that we can buy, sell, copy, and use. As you read through this book, we will introduce several abstract ideas in as concrete a way as possible. As you become familiar and comfortable with these ideas, you will begin to think of the abstractions as actual concrete objects.

Having already gone through this process ourselves, we've chosen to call computer science "the discipline of concrete abstractions"; if that seems too peculiar to fathom, we invite you to read the book and then reconsider the notion. This book is divided into three parts, dealing with procedural abstractions, data abstractions, and abstractions of state. A procedure is a way of abstracting what's called a computational process. Roughly speaking, a process is a dynamic succession of events—a happening. When your computer is busy doing something, a process is going on inside it. When we call a process a computational process, we mean that we are ignoring the physical nature of the process and instead focusing on the information content. [...] What do computer scientists do with processes? First of all, they write descriptions of them. Such descriptions are often written in a particular programming language and are called procedures. These procedures can then be used to make the processes happen. Procedures can also be analyzed to see if they have been correctly written or to predict how long the corresponding processes will take. This analysis can then be used to improve the performance or accuracy of the procedures.

In the second part of the book, we look at various types of data. Data is the information processed by computational processes, not only the externally visible information, but also the internal information structures used within the processes. First, we explore exactly what we mean by the term data, concentrating on how we use data and what we can do with it. Then we consider various ways of gluing small pieces of atomic data (such as words) into larger, compound pieces of data (such as sentences). Because of our computational viewpoint, we write procedures to manipulate our data, and so we analyze how the structure of the data affects the processes that manipulate it. We describe some common data structures that are used in the discipline, and show how to allow disparate structures to be operated on uniformly in a mix-and-match fashion. We end this part of the book by looking at programs in a programming language as data structures. That way, carrying out the computational processes that a program describes is itself a process operating on a data structure, namely the program.

We start the third part of the book by looking at computational processes from the perspective of the computer performing the computation. This shows how procedurally described computations actually come to life, and it also naturally calls attention to the computer's memory, and hence to the main topic of this part, state. State is anything that can be changed by one part of a computation in order to have an effect on a later part of the computation. We show several important uses for state: making processes model real-world phenomena more naturally, making processes that are more efficient than without state, and making certain programs divide into modules focused on separate concerns more cleanly. We combine the new material on state with the prior material on procedural and data abstraction to present object-oriented programming, an approach to constructing highly modular programs with state. Finally, we use the objects' state to mediate interactions between concurrently active subprocesses.

In summary, this book is designed to introduce you to how computer scientists think and work. We assume that as a reader, you become actively involved in reading and that you like to play with things. [...] Our major emphasis is on how computer scientists think, as opposed to what they think about. Our applications and examples are chosen to illustrate various problem-solving strategies, to introduce some of the major themes in the discipline, and to give you a good feel for the subject. We use sidebars to expand on various topics in computer science, to give some historical background, and to describe some of the ethical issues that arise.

Introduzione all'informatica: approccio funzionale / 2

Hal Abelson, Jerry Sussman, Julie Sussman, "Structure and Interpretation of Computer Programs", 1996.

`http://mitpress.mit.edu/sicp/`

Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.

Our goal is that students who complete this subject should have a good feel for the elements of style and the aesthetics of programming. They should have command of the major techniques for controlling complexity in a large system. They should be capable of reading a 50-page-long program, if it is written in an exemplary style. They should know what not to read, and what they need not understand at any moment. They should feel secure about modifying a program, retaining the spirit and style of the original author.

These skills are by no means unique to computer programming. The techniques we teach and draw upon are common to all of engineering design. We control complexity by building abstractions that hide details when appropriate. We control complexity by establishing conventional interfaces that enable us to construct systems by combining standard, well-understood pieces in a "mix and match" way. We control complexity by establishing new languages for describing a design, each of which emphasizes particular aspects of the design and deemphasizes others.

Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."

Introduzione all'informatica: approccio funzionale / 3

Brian Harvey, "Computer Science Logo Style", 1997.

`http://www.cs.berkeley.edu/~bh/v1-toc2.html`

It's quite true that many jobs in the future will involve *using* computers, but the computers will be disguised. When you use a microwave oven, drive a recently built car, or play a video game, you're using a computer, but you didn't have to take a "computer literacy" course to learn how. Even a computer that looks like a computer, as in a word processing system, can be mastered in an hour or two. This book is for people who are interested in computer programming because it's fun.

The Intellectual Content of Computer Programming. [...] the good news is that intellectually serious computer science is within the reach of just about everyone. The bad news is that the curricula tend to be imitations of what is taught to beginning undergraduate computer science majors, and I think that's too rigid a starting point for independent learners, and especially for teenagers. See, the wonderful thing about computer programming is that it *is* fun, perhaps not for everyone, but for very many people. There aren't many mathematical activities that appeal so spontaneously. Kids get caught up in the excitement of programming, in the same way that other kids (or maybe the same ones) get caught up in acting, in sports, in journalism (provided the paper isn't run by teachers), or in ham radio. If schools get too serious about computer science, that spontaneous excitement can be lost. I once heard a high school teacher say proudly that kids used to hang out in his computer lab at all hours, but since they introduced the computer science curriculum, the kids don't want to program so much because they've learned that programming is just a means to the end of understanding the curriculum. No! The ideas of computer science are a means to the end of getting computers to do what you want.

Computer Science Apprenticeship. My goal [...] is to make the goals and methods of a serious computer scientist accessible, at an introductory level, to people who are interested in computer programming but are not computer science majors. If you're an adult or teenaged hobbyist, or a teacher who wants to use the computer as an educational tool, you're definitely part of this audience. [...] I think that for most people programming as job training is nonsense. But if you happen to be interested in programming, studying it in some depth can be valuable for the same reasons that other people benefit from acting, music, or being a news reporter: it's a kind of intellectual apprenticeship. You're learning the discipline of serious thinking and of taking pride in your work. In the case of computer programming, in particular, what you're learning is *mathematical* thinking, or *formal* thinking. (If you like programming, but you hate mathematics, don't panic. In that case it's not really mathematics you hate, it's school. The programming you enjoy is much more like real mathematics than the stuff you get in most high school math classes.) In these books I try to encourage this sort of formal thinking by discussing programming in terms of general rules rather than as a bag of tricks. When I wrote the first edition of this book, in 1984, it was controversial to suggest that not everyone has to learn to program. [...] Today it's more common that I have to fight the opposite battle, trying to convince people why *anyone* should learn about computer programming. After all, there is all that great software out there; instead of wasting time on programming, I'm told, kids should learn to use Microsoft Word or Adobe Illustrator or Macromind Director. At the same time, kids who've grown up with intricate and beautifully illustrated video games are frustrated by the relatively primitive results of their own first efforts at programming. A decade ago it was thrilling to be able to draw a square on a computer screen; today you can do that with two clicks of a mouse. [...] I think a[n] important reason [to learn to program] is that programming—learning how to express a method for solving a problem in a formal language—can still be very empowering. [...]

Why Logo? Logo has been the victim of its own success in the elementary schools. It has acquired a reputation as a trivial language for babies. Why, then, do I use it as the basis for a series of books about serious computer science? Why not Pascal or C++ instead? The truth is that Logo is one of the most powerful programming language available for home computers. [...] It is a dialect of Lisp, the language used in the most advanced research projects in computer science, and especially in artificial intelligence. Until recently, all of the *books* about Logo have been pretty trivial, and they tend to underscore the point by strewing cute pictures of turtles around. [...] the power of a language is a way of measuring how much the language helps you concentrate on the actual problem you wanted to solve in the first place, rather than having to worry about the constraints of the language. [...] In Logo there is only one syntax, the one that invokes a procedure. [...] More powerful languages are based on some particular mathematical model of computing and use that model in a consistent way. For example, APL is based on the idea of matrix manipulation; Prolog is based on predicate calculus, a form of mathematical logic. Logo, like Lisp, is based on the idea of composition of functions. [...] Another dialect, Scheme, has become popular in education. Scheme has many virtues in its own right, but its popularity is also due in part to the fact that it's the language used in the best computer science book ever written: *Structure and Interpretation of Computer Programs*, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. [...] The Scheme approach is definitely more powerful and cleaner for writing advanced projects. Its cost is that the Scheme learner must come to terms from the beginning with the difficult idea of function as object. Logo is more of a compromise with the traditional, sequential programming style. That traditional style is limiting, in the end, but people seem to find it more natural at first. My guess is that ultimately, Logo programmers who maintain their interest in computing will want to learn Scheme, but that there's still a place for Logo as a more informal starting point. [...]

Introduzione all'informatica: approccio orientato agli oggetti / 1

Wanda Dann, Steve Cooper, and Randy Pausch, "Learning to Program with Alice", Prentice Hall, 2006.

`http://www.aliceprogramming.net/`

[*The focus of the Alice project is now to provide the best possible first exposure to programming for students ranging from middle schoolers to college students.*]

This book and the associated Alice system take an innovative approach to teaching introductory programming. There have been relatively few innovations in the teaching of programming in the last 30 years, despite the fact that introductory programming courses are often extremely frustrating to students. The goal of our innovative approach is to allow traditional programming concepts to be more easily taught and more readily understood. The Alice system is free and is available at www.alice.org.

What should a programming course teach?

While many people have strong opinions on this topic, we feel there is a strong consensus that a student in a programming course should learn:

- Algorithmic thinking and expression: being able to read and write in a formal language.
- Abstraction: learning how to communicate complex ideas simply, and to decompose problems logically.
- Appreciation of elegance: realizing that although there are many ways to solve a problem, some are inherently better than others.

What is different about our approach?

Our approach allows students to author on-screen movies and games, where the concept of an "object" is made tangible via on-screen objects that populate a three-dimensional micro world. Students create programs by dragging and dropping program elements (if/then statements, loops, variables, etc.) in a mouse-based editor that prohibits syntax errors. The Alice system is a powerful, modern programming environment that supports methods, functions, variables, parameters, recursion, arrays, and events. We use this strong visual environment to support either an objects-first or an objects-early approach (described in the ACM and IEEE-CS Computing Curricula 2001 report) with an early introduction to events. In Alice, every object *is* an object that students can visibly see! We begin with objects in the very first chapter. In our opinion, there are four primary obstacles to introductory programming:

1. The fragile mechanics, particularly syntax, of program creation

The Alice editing environment removes the frustration of syntax errors in program creation, and allows students to develop an intuition for syntax, because every time a program element is dragged into the editor, all valid "drop targets" are highlighted.

2. The inability to see the results of computation as the program runs

Although textual debuggers and variable watchers are better than nothing, the Alice approach makes the state of the program inherently visible. In a sense, we offload the mental effort from the student's cognitive system to his or her perceptual system. It is much easier for a student to see that an object has moved backward instead of forward, as opposed to noticing that the "sum" variable has been decremented, rather than incremented. Alice's visual nature allows students to see how their animated programs run, affording an easy relationship of the program construct to the animation action. Today's students are immersed in a world where interactive, three-dimensional graphics are commonplace; we try to leverage that fact without pandering to them.

3. The lack of motivation for programming

Many students take introductory programming courses only because they are required to do so. Nothing will ever be more motivating than a stellar teacher, but the right environment can go a long way. In pilot studies of classes using Alice, students do more optional exercises and are more likely to take a second class in programming than control groups of students using traditional tools. The most common request we received regarding earlier versions of Alice was the ability to share creations with peers; we have added the ability to run Alice programs in a WWW browser so students can post them on their web pages. Although we have seen increased motivation for all students, we have seen especially encouraging results with underrepresented student groups, especially female students.

4. The difficulty of understanding compound logic and learning design techniques

The Alice environment physically encourages the creation of small methods and functions. More importantly, the analogy of making a movie allows us to utilize the concept of a storyboard, which students know is an established movie-making process. We illustrate design techniques using simple sketches and screen captures. And, we encourage the use of textual storyboards, which are progressively refined, essentially designing with pseudocode.

Introduzione all'informatica: approccio orientato agli oggetti / 2

Joseph Bergin, Mark Stehlik, Jim Roberts, Richard Pattis, "Karel J. Robot – A Gentle Introduction to the Art of Object-Oriented Programming in Java", 2005.

`http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html`

The programming landscape has changed significantly since the initial publication of Karel the Robot in 1981. Today there are new programming languages, new programming paradigms, and new and more powerful computers. Pascal no longer enjoys the popularity it did in the 1980s. [...] However, the object-oriented programming paradigm has begun to dominate the world of commercial software production. Karel J. Robot updates Karel the Robot [...] to provide a means of introducing novice programmers to object-oriented programming (OOP). This book [...] provides instruction that is thoroughly object-oriented from the beginning. Where the original used a syntax and methodology derived from Pascal, the present text is based closely on Java. In object-oriented programming, a computation is carried out by a set of interacting objects. Here, the objects are robots that exist in a simple world. There can be one or several robots assigned to a task. The programming task is divided into two parts. The first part is defining the capabilities of the robots that are needed. The second is providing a description of the task for the robots to perform. The programmer uses his or her problem solving skills on both parts of this task. This version [... puts] more emphasis on polymorphism, the primary distinction between procedural programming and object-oriented programming. [...]

We believe that most people will not actually have to program a computer as part of their everyday lives, either now or in the future. However, many people will need to be able to use a computer and will occasionally need to do something with the machine beyond the "ordinary." [...] This book will introduce you to problem-solving approaches that can be used with computers. Unfortunately, some people believe programming requires a "different" way of thinking. We don't agree with this statement. Instead of changing the way you think, this book will change how you apply your problem-solving skills to different kinds of problems. The original Karel the Robot used procedures as the fundamental problem solving medium, as is appropriate in procedural programming. Here we apply our problem solving skills, instead, to the design of classes that describe objects (robots), as classes are the primary means of breaking a complex problem into manageable parts in object-oriented programming. The skills of procedural programming and object-oriented programming are very similar, though we look at problems from a slightly different perspective when using OOP. [...] For novice programmers, this book will give some insight into the programming process from two distinctly different points of view: the planner's and the implementer's. All the problems can be thought about, discussed, and planned in English. Once you have developed your plan, the actual syntax of the robot programming language has very few rules to get in your way as you become the implementer or programmer. [...]

[...]

We (the authors) have left out nearly everything about Java primitive data (int…) because including it complicates the student's life and doesn't teach them about OO. There is one computational model: message passing. State is implicit in the world. In this regard, [Karel J Robot] is much like a functional system. The students don't program with explicit state and variables. The book doesn't depend on the von Neumann architecture. Objects *do* things and they *remember* things, just like people. The only variables are reference variables (other than what is needed to support the for loop). We use these variables to "deliver robots" and to set up delegation (a big topic) [...].

The overall idea of this book and its simulator is actually applicable to all computer programming. [...] *When you are faced with a problem to solve, don't just solve it in the language at hand. First design a new language in which the original problem would be easy to solve. Then implement the language in the one at hand and finally solve the original problem in the new language*. Karel J Robot is a language in this sense. The problem it tries to solve is how to teach students computer problem solving in a compelling way while moving them in the direction they really need to go. Building a robot (or other) class changes the language. Since we really use only the language of message passing (not manipulation of primitives, etc), there is one computational model and so a new class really extends that language. So, writing a class raises the level of the language toward that of the problem, but without changing or complicating the syntax. Then you solve the problem.

Throughout the book we depend on metaphor to help us teach. I don't try to show how this stuff is implemented on a lower level machine. Instead, I relate it to what the student already knows about the real world. [...] The main guiding metaphor is objects (like people) *do* things and they *remember* things. Like people, they are autonomous and control their own actions (polymorphism). The flaw in the metaphor is that with people message passing is two-way. With objects it is one-way unless you explicitly set up the other direction. [...] The book is actually the first cycle of a spiral learning approach to programming. None of the topics is covered exhaustively, but each is done enough to enable serious problem solving. After you finish the book, you take up other topics with your students, but return to these in deeper detail. [...]

Introduzione all'informatica: approccio orientato agli oggetti / 3

David J. Barnes, Michael Kölling, "Objects First with Java", 2006.

`http://www.bluej.org/objects-first/`

[...] The main focus of the book is general object-oriented and programming concepts from a software engineering perspective.

[...]

Real objects first

One of the reasons for choosing BlueJ was that it allows an approach where teachers truly deal with the important concepts first. 'Objects early' has been a battle cry for many textbook authors and teachers for some time. Unfortunately, the Java language does not make this noble goal very easy. Numerous hurdles of syntax and detail have to be overcome before the first experience with a living object arises. The minimal Java program to create and call an object typically includes:

- writing a class;
- writing a main method, including concepts such as static methods, parameters, and arrays in the signature;
- a statement to create the object ('new');
- an assignment to a variable;
- the variable declaration, including variable type;
- a method call, using dot notation;
- possibly a parameter list.

[...]

With BlueJ, this is not a problem. [...]

[...]

An iterative approach

Another important aspect of this book is that it follows an iterative style. In the computing education community, a well-known educational design pattern exists that states that important concepts should be taught early and often (The "Early Bird" pattern, in J. Bergin: *"Fourteen Pedagogical Patterns for Teaching Computer Science"*, [...]). It is very tempting for textbook authors to try and say everything about a topic at the point where it is introduced. For example, it is common, when introducing types, to give a full list of built-in data types, or to discuss all available kinds of loops when introducing the concept of a loop.

These two approaches conflict: we cannot concentrate on discussing important concepts first, and at the same time provide complete coverage of all topics encountered. Our experience with textbooks is that much of the detail is initially distracting, and has the effect of drowning the important points, thus making them harder to grasp.

In this book, we touch on all of the important topics several times, both within the same chapter and across different chapters. Concepts are usually introduced at a level of detail necessary for understanding and applying the task at hand. They are revisited later in a different context, and understanding deepens as the reader continues through the chapters. This approach also helps to deal with the frequent occurrence of mutual dependencies between concepts.

Some teachers may not be familiar with an iterative approach. Looking at the first few chapters, teachers used to a more sequential introduction will be surprised about the number of concepts touched on this early. It may seem like a steep learning curve.

It is important to understand that this is not the end of the story. Students are not expected to understand everything about these concepts immediately. Instead, these fundamental concepts will be revisited again and again throughout the book, allowing students to get a deeper and deeper understanding over time. Since their knowledge level changes as they work their way forward, revisiting important topics later allows them to gain a deeper understanding overall.

Introduzione all'informatica: approccio orientato agli oggetti / 4

Lynn Stein, "Interactive Programming in Java", 2003.

`http://www.cs101.org/ipij/`

[...] It is the first textbook to rethink the traditional curriculum in light of the current interaction-based computer revolution. *Interactive Programming* shifts the foundation on which the teaching of Computer Science is based, treating computation as *interaction* rather than *calculation* [...].

Traditionally, introductory programming teaches algorithmic problem-solving. In this view, a program is a sequence of instructions that describe the steps necessary to achieve a desired result. The 'pieces' of this program are these steps. They are combined by sequencing. The program produced is evaluated by means of its end result. Students trained in this way often have difficulty moving beyond the notion that there is a single thread of control over which they have complete control. In contrast, most programs of interest today are made up of implicitly or explicitly concurrent components that interact to provide ongoing services. Buzzwords such as "client/server" and "event-driven" are part of the descriptive language of this new generation of programs. Embedded systems and software agents typify their incarnations. User interface design, distributed programming, and the world-wide web are logical extensions of a way of thinking that has interaction at its core. When programming is taught from a traditional perspective, important topics like these are treated as advanced and inaccessible to the introductory student. It is unsurprising that senior software engineers report that today's undergraduates are ill-equipped to handle the realities of embedded interactive software. Most require on-the-job retraining to "think concurrently." Students trained in the traditional curriculum are often so indoctrinated in the "sequence of steps" mentality that they can no longer rely on the intuition common to every child coordinating a group of friends or trying to sneak a cookie behind her parent's back.

Interactive Programming provides an alternate entry into the computer science curriculum. It teaches problem decomposition, program design, construction, and evaluation, beginning with the following premises: A program is a community of interacting entities. Its "pieces" are these implicitly or explicitly concurrent entities: user interfaces, databases, network services, etc. They are combined by virtue of ongoing interactions which are constrained by interfaces and by protocols. A program is evaluated by its adherence to a set of invariants, constraints, and service guarantees—timely response, no memory leaks, etc. Because it begins from this alternate notion of what programming is about, *Interactive Programming* tells a rather different story from the traditional introductory programming book. By its end, students are empowered to write and read code for client-server chat programs, networked video games, web servers, user interfaces, and remote interaction protocols. They build event-driven graphical user interfaces and spawn cooperating threads. Each of these programs—ll of which are beyond the scope of traditionally taught introductory courses—is a natural extension of the community metaphor for computation.

Many computer science departments are contemplating a change to the Java programming language for introductory computer science courses. While it is possible to make this change without transforming the introductory curriculum, adopting Java without a corresponding curricular change amounts to sweeping more and more of what is important in today's computational world under the rug. Java embodies much of modern programming practice. Insisting on traditional approaches actually makes certain aspects of the language less accessible. Shifting to a curriculum in which concurrent interacting entities play a central role makes far more of modern computation theory, practice, and tools accessible to today's introductory student.

Introduzione all'informatica: approccio a spirale (breadth-first) / 1

Carl Burch, "The science of computing", 2004.

`http://www.cburch.com/socs/`

Computer science is the study of algorithms for transforming information. In this course, we explore a variety of approaches to one of the most fundamental questions of computer science: What can computers do? That is, by the course's end, you should have a greater understanding of what computers can and cannot do. We frequently use the term computational power to refer to the range of computation a device can accomplish. Don't let the word power here mislead you: We're not interested in large, expensive, fast equipment. We want to understand the extent of what computers can accomplish, whatever their efficiency. [...]

Common misconceptions about computer science

Most students arrive to college without a good understanding of computer science. Often, these students choose to study computer science based on their misconceptions of the subject. In the worst cases, students continue for several semesters before they realize that they have no interest in computer science. Before we continue, let me point out some of the most common misconceptions.

Computer science is not primarily about applying computer technology to business needs. Many colleges have such a major called "Management Information Systems," closely related to a Management or Business major. Computer science, on the other hand, tends to take a scientist's view: We are interested in studying computation in itself. When we do study business applications of computers, the concentration is on the techniques underlying the software. Learning how to use the software effectively in practice receives much less emphasis.

Computer science is not primarily about building faster, better computers. Many colleges have such a major called "Computer Engineering," closely related to an Electrical Engineering major. Although computer science entails some study of computer hardware, it focuses more on computer software design, theoretical limits of computation, and human and social factors of computers.

Computer science is not primarily about writing computer programs. Computer science students learn how to write computer programs early in the curriculum, but the emphasis is not present in the "real" computer science courses later in the curriculum. These more advanced courses often depend on the understanding built up by learning how to program, but they rarely strive primarily to build programming expertise.

Computer science does not prepare students for jobs. That is, a good computer science curriculum isn't designed with any particular career in mind. Often, however, businesses look for graduates who have studied computer science extensively in college, because students of the discipline often develop skills and ways of thinking that work well for these jobs. Usually, these businesses want people who can work with others to understand how to use computer technology more effectively. Although this often involves programming computers, it also often does not.

Thinking about your future career is important, but people often concentrate too much on the quantifiable characteristics of hiring probabilities and salary. More important, however, is whether the career resonates with your interests: If you can't see yourself taking a job where a major in is important, then majoring in isn't going to prove very useful to your career, and it may even be a detriment. Of course, many students choose to major in computer science because of their curiosity, without regard to future careers.

Textbook goals

This textbook fulfills two major goals.

- It satisfies the curiosity of students interested in an overview of practical and theoretical approaches to the study of computation.
- Students who believe they may want to study computer science more intensively can get an overview of the subject to help with their discernment. [...]

The course on which this textbook is based [...] has three major units, of roughly equal size.

- Students learn the fundamentals of how today's electronic computers work [...]. We follow a "bottom-up" approach, beginning with simple circuits and building up toward writing programs for a computer in assembly language.
- Students learn the basics of computer programming, using the specific programming language of Java [...].
- Students study different approaches to exploring the extent of computational power [...].

Introduzione all'informatica: approccio a spirale (breadth-first) / 2

Mark Guzdial, Barbara Ericson, "Introduction to Computing and Programming with Java: A Multimedia Approach", 2007.

`http://coweb.cc.gatech.edu/mediaComp-plan/101`
(vedi anche `http://coweb.cc.gatech.edu/mediaComp-teach`)

The purpose of this book is to introduce computing in a way that students find motivating, creative, and relevant. Students enjoy writing programs to modify pictures, sounds, Web pages, and movies. We still teach all the usual introductory concepts such as variables, methods, arrays, looping, conditionals, objects, classes, inheritance, and interfaces, but in a multimedia context. One of the advantages of this approach is that it is easy to tell if your program is working or not by looking at (or listening to) the resulting media.

This media computation approach was first created for an undergraduate course at Georgia Tech for non-majors using Python. This course has increased the success rate for non-majors (business majors changed from a 49% success rate to an 88% success rate). Other colleges and universities have trialed the Python version with similar results. [...] The Georgia Tech introductory course has resulted in non-major students taking a computer science minor and even caused some to become computer science majors. And, with a 40-60% drop in computer science majors at the college level across the country, it is important that we attract people to computer science, instead of driving them away. We particularly would like to attract more women and minorities [...] to computer science. The speed of modern computers means that we can introduce computer concepts by manipulating media instead of just having the computer print out "Hello World" and other such assignments which we have been using for over 30 years. Students do not find programs that convert temperature or compute sales tax very interesting. They do not find such examples relevant to their lives.

[...]

One of the lessons from the research on computing education is that one doesn't just "learn to program." One learns to program *something*. How motivating that *something* is can make the difference between learning to program or not. Some people are interested in learning programming just for programming's sake—but that's not most people. Unfortunately, most introductory programming books are written as if students have a burning desire to learn to program. They emphasize programming concepts and give little thought to making the problems that are being solved interesting and relevant. They introduce new concepts without showing why the students *should* want to know about them.

In this book students will learn about programming by writing programs to manipulate media. Students will create and modify images, such as correcting for "red-eye" and generating negative images. Students will modify sounds, like splicing words into sentences or reversing sounds to make interesting effects. Students will write programs to generate Web pages from data in databases, in the same way that CNN.com and Amazon.com do. They will create animations and movies using special effects like the ones seen on television and in movies. [...] This book is about teaching people to program in order to communicate. People want to communicate. [...] Increasingly, the computer is used as a tool for communication even more than as a tool for calculation. Virtually all published text, images, sounds, music, and movies today are prepared using computing technology. This book focuses on how to manipulate images, sounds, text, and movies as professionals might, but with programs written by the students. [...] The last chapters at the end of the book are about *computing*, not just programming. The computer is the most amazingly creative device that humans have ever conceived of. It is literally completely made up of mind-stuff. [...] Playing with programming can be and *should* be enormous fun.

The media computation approach used in thi book starts with what students use computers for [...] We then explain programming and computing in terms of these activities. [...] The approach in this book is different than in many introductory programming books. We teach the same computing concepts but not necessarily in the usual order. For example, while we create and use objects early, we don't have students defining new classes till fairly late. Research in computing education suggests that learning to program is hard and that students often have trouble with the basics (variables, iteration, and conditionals). [...] We introduce new concepts only after setting the stage for why we would need them. [...] We repeat concepts in different media to increase the odds that students will find an explanation and relevance that works for them, or better yet, find *two* or more explanations that work for them. The famous artificial intelligence researcher Marvin Minsky once said that if you understand something in only one way, you don't understand it at all. Repeating a concept in different relevant settings can be a powerful way of developing flexible understandings. Memory is associative [...]. People can learn concepts and skills on the promise that it will be useful some day, but the concepts and skills will be rrelated only to the promises, not to everyday life. [...] If we want students to gain *transferable* knowledge [...], we have to help them to relate the knowledge to more general problems [...]. Thus, we teach with concrete experiences that students can explore and relate to [...].

Introduzione all'informatica: approccio a spirale (breadth-first) / 3

Alan W. Biermann, Dietolf Ramm, "Great Ideas in Computer Science with Java", 2001.

`http://vig.pearsoned.co.uk/catalog/academic/product/0,1144,0321434455,00.html`
(vedi anche `http://www.cs.duke.edu/~awb/GICS/`)

This book presents the "great ideas" of computer science, condensing a large amount of complex material into a manageable, accessible form; it does so using the Java programming language. The book is based on the problem-oriented approach that has been so successful in traditional quantitative sciences. For example, the reader learns about database systems by coding one in Java, about system architecture by reading and writing programs in assembly language, about compilation by hand-compiling Java statements into assembly language, and about noncomputability by studying a proof of noncomputability and learning to classify problems as either computable or noncomputable. The book covers an unusually broad range of material at a surprisingly deep level. It also includes chapters on networking and security. Even the reader who pursues computer science no further will acquire an understanding of the conceptual structure of computing and information technology that every well-informed citizen should have.

Table of Contents
- Studying Academic Computer Science: An Introduction
- The World Wide Web
- Watch Out: Here Comes Java
- Numerical Computation and a Study of Functions
- Top-Down Programming, Subroutines, and a Database Application
- Graphics, Classes, and Objects
- Simulation
- Software Engineering
- Machine Architecture
- Language Translation
- Virtual Environments for Computing
- Security, Privacy, and Wishful thinking
- Computer Communications
- Program Execution Time
- Parallel Computation
- Noncomputability
- Artificial Intelligence

Introduzione all'informatica: approccio basato sull'hardware

Yale N. Patt, Sanjay J. Patel, "Introduction to Computing Systems : from bits and gates to C and beyond", 2004.

`http://highered.mcgraw-hill.com/sites/0072467509/`

This textbook has evolved from EECS 100, the first computing course for computer science, computer engineering, and electrical engineering majors at the University of Michigan, that Kevin Compton and the first author introduced for the first time in the fall term, 1995. EECS 100 happened because Computer Science and Engineering faculty had been dissatisfied for many years with the lack of student comprehension of some very basic concepts. For example, students had a lot of trouble with pointer variables. Recursion seemed to be "magic," beyond understanding. We decided in 1993 that the conventional wisdom of starting with a high-level programming language, which was the way we (and most universities) were doing it, had its shortcomings. We decided that the reason students were not getting it was that they were forced to memorize technical details when they did not understand the basic underpinnings. The result is the bottom-up approach taken in this book.

We treat (in order) MOS transistors (very briefly, long enough for students to grasp their global switch-level behavior), logic gates, latches, logic structures (MUX, Decoder, Adder, gated latches), finally culminating in an implementation of memory. From there, we move on to the Von Neumann model of execution, then a simple computer (the LC-2), machine language programming of the LC-2, assembly language programming of the LC-2, the high level language C, recursion, pointers, arrays, and finally some elementary data structures. We do not endorse today's popular information hiding approach when it comes to learning. Information hiding is a useful productivity enhancement technique after one understands what is going on. But until one gets to that point, we insist that information hiding gets in the way of understanding. Thus, we continually build on what has gone before, so that nothing is magic, and everything can be tied to the foundation that has already been laid.

We should point out that we do not disagree with the notion of top-down design. On the contrary, we believe strongly that top-down design is correct design. But there is a clear difference between how one approaches a design problem (after one understands the underlying building blocks), and what it takes to get to the point where one does understand the building blocks. In short, we believe in top-down design, but bottom-up learning for understanding.

Introduzione all'informatica: approccio algoritmico

Russel L. Shackelford, "Introduction to Computing and Algorithms", 1998.

`http://portal.acm.org/citation.cfm?id=550299`

Description
This book prepares students for the world of computing by giving them a solid foundation in the "science" of computer science—algorithms. By taking an algorithm-based approach to the subject, this new book helps readers grasp overall concepts, rather than getting them bogged down with specific syntax details of a programming language that can become obsolete quickly. By working with algorithms from the start and applying the concepts to the real world, students will understand the power of computers as problem solving tools and learn to think like programmers.

Table of Contents
I. The Computing Perspective.
- Technology, Science and Culture.
- The Algorithmic Model.

II. The Algorithm Toolkit.
- Basic Data, Operations, and Decisions.
- Tools for "Procedural Abstraction."
- Tools for "Data Abstraction."
- Algorithmic Methods.
- Tools for Estimating Cost and Complexity.
- Tools for Verifying Correctness.
- Tools for "Behavioral Abstraction."

III. The Limits of Computing.
- Concurrency and Parallelism.
- Hierarchies of Complexity.

Introduzione all'informatica: approccio interdisciplinare

Robert Sedgewick, Kevin Wayne, "Introduction to Programming in Java: An Interdisciplinary Approach", 2007.

`https://introcs.cs.princeton.edu/java/home/`

[...] Learning to program is an essential part of the education of every student in the sciences and engineering. Beyond direct applications, it is the first step in understanding the nature of computer science's undeniable impact on the modern world. This book aims to teach programming to those who need or want to learn it, in a scientific context.

Our primary goal is to *empower* students by supplying the experience and basic tools necessary to use computation effectively. Our approach is to teach students that writing a program is a natural, satisfying, and creative experience (not an onerous task reserved for experts). We progressively introduce essential concepts, embrace classic applications from applied mathematics and the sciences to illustrate the concepts, and provide opportunities for students to write programs to solve engaging problems.

We use the Java programming language for all of the programs in this book — we refer to Java after programming in the title to emphasize the idea that the book is about *fundamental concepts in programming*, not Java per se. This book teaches basic skills for computational problem-solving that are applicable in many modern computing environments, and is a self-contained treatment intended for people with no previous experience in programming.

This book is an *interdisciplinary* approach to the traditional CS1 curriculum, where we highlight the role of computing in other disciplines, from materials science to genomics to astrophysics to network systems. This approach emphasizes for students the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world. While it is a CS1 textbook designed for any first-year college student interested in mathematics, science, or engineering (including computer science), the book also can be used for self-study or as a supplement in a course that integrates programming with another field.

Coverage
The book is organized around four stages of learning to program: basic elements, functions, object-oriented programming, and algorithms (with data structures). We provide the basic information readers need to build confidence in writing programs at each level before moving to the next level. An essential feature of our approach is to use example programs that solve intriguing problems, supported with exercises ranging from self-study drills to challenging problems that call for creative solutions.

[...]

Applications in science and engineering are a key feature of the text. We motivate each programming concept that we address by examining its impact on specific applications. We draw examples from applied mathematics, the physical and biological sciences, and computer science itself, and include simulation of physical systems, numerical methods, data visualization, sound synthesis, image processing, financial simulation, and information technology. Specific examples include a treatment in the first chapter of Markov chains for web page ranks and case studies that address the percolation problem, N-body simulation, and the small-world phenomenon. These applications are an integral part of the text. They engage students in the material, illustrate the importance of the programming concepts, and provide persuasive evidence of the critical role played by computation in modern science and engineering.

Our primary goal is to teach the specific mechanisms and skills that are needed to develop effective solutions to any programming problem. We work with complete Java programs and encourage readers to use them. We focus on programming by individuals, not library programming or programming in the large (which we treat briefly in an appendix).

Contents

Elements of Programming
- Your First Program
- Built-in Types of Data
- Conditionals and Loops
- Arrays
- Input and Output
- Case Study: Random Web Surfer

Functions and Modules
- Static Methods
- Libraries and Clients
- Recursion
- Case Study: Percolation

Object-Oriented Programming
- Data Types
- Creating Data Types
- Designing Data Types
- Case Study: N-body Simulation

Algorithms and Data Structures
- Performance
- Sorting and Searching
- Stacks and Queues
- Symbol Tables
- Case Study: Small World

Introduzione all'informatica: programmazione logica?

A questo riguardo riporto solo due estratti che individuano alcune difficoltà nell'introdurre la programmazione logica.

Paul Brna, "Logic Programming in Education: a Perspective on the State of the Art", in *Proceedings of the Post-Conference Workshop on Logic Programming and Education* (Bottino, Forcheri, and Molfino Eds.), 1994.

The interest in the use of Logic Programming in education stems from the great enthusiasm for Prolog that arose in the late seventies. However, Prolog is neither the purest Logic Programming language nor the easiest language for novices to come to terms with—but the opportunity to introduce very powerful ideas by teaching students to program in Prolog has been seized by many teachers at all levels of education.

[...]

Prolog's backtracking behaviour causes problems for novices. [... Taylor's] analysis indicated different patterns of behaviour amongst students as to whether they were using declarative or procedural views of Prolog. Both patterns extend the issues associated with teaching Prolog to areas beyond ones connected with teaching some consistent interpretation of the Prolog interpreter. She identified ways in which students avoid following Prolog execution. [... Empirical] work supports the notion that there is a stage of development during which many students possess both multiple misconceptions and competing models of Prolog search control. We know relatively little about the ways in which Prolog expertise develops. [...] Prolog unification is an area which holds many difficulties for novice programmers [...]. The reasons for these difficulties may vary: the student may hold relatively stable misconceptions; be uncertain about the necessary knowledge; or possess correct, but incomplete knowledge. Alternatively, the particular problem may place too great a demand on short-term memory resources with the result that the student produces slips in performance. Whatever the cause of the problem, it would seem that unification doesn't remain a problem for very long. However, when the novice is still learning Prolog syntax and elementary notions of control flow the additional problems of learning about unification can be severe.

J. Taylor, "Analyzing Novices Analyzing Prolog: What stories do novices tell themselves about Prolog?", in *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study* (Brna, du Boulay, and Pain Eds.), Ablex Publishing Co., 1999.

Informal observations of Prolog learners showed that, despite being presented with correct information and models, students still tended to construct their own idiosyncratic explanations of events, and, characteristically, they defended these 'stories' fiercely when tutorial intervention was attempted. Although the stories were often so flawed that the student's future progress was potentially hampered, it was nevertheless true that learning could not have proceeded *at all* without them. [...] Observational studies were undertaken which showed that students used their tacit knowledge of human discourse processes both to interpret the *language* used to communicate with the computer and to interpret the *behaviour* of the machine. Students did not appreciate the fundamental differences between natural discourse (as takes place amongst humans) and formal discourse (as takes place between humans and machines), and confused elements of the discourse levels. This can be an effective initial learning strategy, but unless its limitations are recognised, programs are inevitably incomplete at some level.

M. Clancy, "Misconceptions and attitudes that interfere with learning to program", in *Computer Science Education Research* (S.A. Fincher and M. Petre Eds.) – Part II, Ch. 1, Routledge, 2004.

PROLOG è stato insegnato in corsi iontroduttivi [...] fin dall'inizio degli anni '80. [...]
Le operazioni di unificazione [...] e depth-first search [...] sono fornite come primitive in PROLOG. Queste sono tanto potenti quanto difficili da capire, anche per programmatori esperti. [...]
Non deve quindi sorprendere che gli studenti che apprendono PROLOG sviluppino una quantità di misconcezioni.