The HOP! project: toward teaching programming for all

Henri Lesourd

The HOP! project: toward teaching programming for all

Abstract. Because, to a large extent, programming involves many *implicit* skills that can only be acquired through *practice*, the teaching of programming still appears to be unsatisfactory. In this paper, we present the HOP! system (aka. Hands-On-Programming!), which is an interactive programming environment intended to be used to teach programming by means of incrementally developed *examples*, starting from simple projects to move gradually towards more complicated achievements. The philosophy and design of the HOP! system stem from a number of recent developments in the fields of linguistics, psychology and pedagogy, as well as from an analysis of the *problem solving* activity in programming. The HOP! system is freely distributed via the Internet (see http://www.hop-project.org).

Because, to a large extent, programming involves many *implicit* skills that can only be acquired through *practice*[1], the teaching of programming should in our opinion be based on the development of projects, which should ideally be as varied as possible. In this paper, we present the HOP! system (aka. Hands-On-Programming!), which is an interactive programming environment intended to be used to teach programming by means of incrementally developed *examples*, starting from simple projects to move gradually towards more complicated achievements. First of all, we examine the history and evolution of computers from the 50s to today (part **1**); we then analyze how the ideas which underlie the development of the modern industrial societies in which we live have been built (section **2.1**), and we also consider a number of recent developments in the fields of linguistics, psychology and pedagogy (section **2.2** and **2.3**), as well as some of the reasons why freedom is of vital importance for the development of the mind; we then draw a number of conclusions about the method, as well as about the spirit in which the teachings related to science and technology should in our view be conducted (section **2.4**). Arrived here, we analyze the *problem solving* activity in programming (part **3**). Finally, we offer a synthesis, and we present the choices made for the system design, along with the experience gained and the developments we envision today for the future developments of the HOP! project (parts **4** and **5**).

1 The evolution of computing, from the 50s to today

1.1 Discovery of concepts and experimental phase

Calculating machines date back to antiquity (e.g. mechanism of Antikythera [FRE06], abacus), but the programmable machine is invented in the XIXth century (Babbage machine), and it is formalized and implemented in the following century by Turing (which develops one among the fundamental mathematical models of the computer, the *Turing machine*, and works for the Allied war effort, in particular to develop the "bombs" which have been used to break the German codes during World war II[2]) and by Von Neumann, who brings the concept to its modern form.

1. We not mean that theoretical knowledge is not important, on the contrary, the problem is that without practice, the theory is useless and can not in fact be truly understood. Conversely, when it is not sufficiently informed by the state of the art, the practice runs the risk of being sooner or later confronted with insurmountable conceptual obstacles.

Over the next 30 years (i.e., roughly from the 50s to the 80s), various kinds of computers are developed, and the basic concepts of computer architecture, operating systems and software development gradually appear. Especially:

- Computer programming, which was initially performed in machine language (i.e. directly by means of switches used to encode binary numbers representing the instructions) evolves, and ways to program by means of *symbolic* codes are developed. It is the invention of *programming languages*, which begins with the first assembly languages (which are languages that allow writing machine language using symbols that a special program, the *assembler*, automatically encodes in the appropriate way), and continues with algorithmic languages, through which it becomes possible to express the required calculations by means of mathematical formulas and complex instructions (one of the first languages of this kind is Fortran, i.e., FORmula TRANslator), which are broken down into simple elements by an assembler of a new kind, called a *compiler*. A milestone is reached when, following the pioneering work of Gödel in the 30s (who, to develop the proof of his incompleteness theorems, uses an *encoding* of logical formulas by means of large numbers), programming languages having features allowing the manipulation of non-numeric data (e.g. strings, as in Snobol [GRI75] or sets, as in Setl [SCH86], as well as lists, as in Lisp [MCC60]) are finally developed.

  A number of formal techniques are being developed at that time, especially the *formal grammars* (which have been used for the first time to specify the programming language Algol [NAU63]), along with the first approaches to prove the correctness of programs (e.g. Hoare logic [HOA69]) ;

- To overcome the limitations of operating the computers in batch mode, the first *operating systems* are developed, which make it possible to use the computer in *multiprograming mode*, where several programs are working in a seemingly simultaneous manner, and share the computing power of the computer.

  During this period were developed the key concepts which up to now, underpin the design of operating systems, especially the concept of *protection* (which, by means of a physical segmentation of the memory and of a two-tier instruction system[3], ensures that a user program can never access the memory area allocated to another program or to the code of the operating system [NEE72]), the concepts related to the *synchronization* of parallel tasks, which in particular, consists of making certain series of operations *atomic* [HOA74], along with the concept of a *hierarchical file system*, which appears for the first time in Multics [DAL65], as well as the first *virtual machines*, initially developed in VM/370 [CRE81] ;

- Experience in the development of the first large projects brings out a number of important theoretical concepts, such as the notion of abstract data type [LIS74], and the notions of module and encapsulation [PAR72].

  At that time, there is a growing awareness, and the intrinsic difficulties attached to the development of large software [BRO75], along with the (relative) specificity of the techniques to be used, become a subjected of serious enquiry. The two founding conferences of the *software engineering* discipline are held in the late 60s [NAU69][RAN70]. The *software development cycle* starts to be better understood, especially the problems related to the integration of modules developed independently: the difference between *programming in the small* (development of small modules) and *programming in the large* (cutting down a large project into modules and integration of these modules into a coherent whole) which is then conceptualized remains the core of large software development until today ;

2. See `https://en.wikipedia.org/wiki/Bombe`

3. The usual instructions may be included in user programs; the hardware-related instructions cannot (they are only available in *super user* mode, which is activated to handle critical system resources when needed. The user programs are *delegating* the execution of the tasks that handle critical resources; these tasks are implemented by the operating system programs, which are the only ones able to run in superuser mode, and whose code cannot be changed by users).

- Multiprogramming enables users to work *interactively*, and paves the way for the use of computers as a universal tool that can be used to develop all kinds of works, be it text, images, sound or video, in short, all kinds of activities traditionally developed by means of analog media, which could previously not be processed at all by means of computers[4].

  During this pioneering period are invented: the first *interactive text editors* [IRO72], the first computer-aided design software [SUT63], the first document description languages based on markup (especially TEX [KNU84] and LATEX [LAM86], which revolutionize mathematical document editing), the first *hypertext* systems [ENG62], the first *object-oriented* programming languages [DAH67], and finally the first *windowing* operating systems [GOL89] in which *wysiwyg*[5] interactive edition becomes the default [REE79] ;

- Finally, there is a changeover from the centralized mode of operating to the distributed mode, with the development of *networks* that allow teams spread out in many locations to work together (synchronously or asynchronously) ;

  The first packet switched networks are developed in the 60s [BAR64][ROB78], then appears the TCP/IP protocol [CER74], which unifies the heterogeneous networks in existence at that time, and emerges as the facto standard underlying the development of the Internet. At about the same time, the standards that define the electronic mail appear (e.g. POP [REY84], SMTP [POS81]), and a little bit later, HTTP [BER91], which is the starting point of the World Wide Web (WWW) [BER89] ;

All this represents a considerable amount of *concepts*, which is not likely to be easily rediscovered by a single person (we will come back to this).

Furthermore, it is important to note that even today, these concepts remain the basis underlying the design of the computing systems (even recent) that we use everyday. To believe that in computing technology, "innovations" bring true revolutions every six months or even every two years, amounts to ignore the actual historical perspective (for example, Ben-Ari [BAR05] emphasizes that the Java programming language, appeared in the 90s, does not actually bring any radical innovation, since it is built on concepts which date back from the 70s at the latest). What evolves often quite rapidly are the "technologies" which are, as a matter of fact, *products* which, one after the other, reach a dominant market position. But beyond the hype, the actual evolution of the underlying concepts is *slow* ; in our opinion, it is these concepts that one should try to understand, rather than their umpteenth variation as it appears in the most hyped framework at a given time.

In this pioneering period of discovery and clarification, which ends almost at the end of the 80s (Smalltalk, 1980; the web, 1990), we were having relatively small teams facing relatively circumscribed development problems. By that time, however, the extreme difficulty of verifying software is perceived by some observers (e.g. in 1984, Ken Thompson [THO84], one of the inventors of the UNIX system writes: « The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) »).

Today the situation has not fundamentally changed, and software development techniques remain very difficult to control. In 1986, Brooks [BRO86] writes that it will perhaps always be the case, and to better control the complexity of the processes, he proposes to:

a) Harness market forces to avoid building components that can be purchased ;

b) Use rapid prototyping techniques ;

c) Develop software in an incremental manner ;

d) Find the most talented software designers of a generation, and encourage them to develop their talents ;

This indeed foreshadows a number of developments that have occurred since[6], but which as we shall see, have not solved all the problems.

4. For example, the simple task of counting the number of occurrences of a word in a text: in practice, without a computer, this kind of statistic can easily take weeks, months or even years.

5. WYSIWYG ≡ What You See Is What You Get.

6. In particular, the items (b) and (c) above seem to announce pretty much the "agile" or "extreme programming" kind of development techniques which appeared thereafter.

1.2 Explosion

In the early 90s, still endures a tendency to try to use development methods on a human scale, which in many cases can be substituted with advantage for the heavy infrastructure of big projects (this is the case first of all with UNIX, and with a lot of components of the Internet infrastructure that followed, as well as with many software which still play an important role today, for example TEX, the kernel of the LATEX system, or with a number of medium-sized tools that exist in the world of scientific computing and elsewhere[7], which in one way or another, were developed independently of one another, and were later combined with each other).

On the commercial side, initially, the personal computer is also relatively open and relies on user participation, as was the case in Smalltalk, in personal computers equipped with a Basic interpreter [KEM68], or in HyperCard [LAS12] and Applescript [COO07].

But gradually, the personal computing industry promotes software which is more and more based on the use of metaphors allowing people to use computers without needing to understand how the underlying system (Macintosh, then Windows) works, and banks on the amount of lines of code, as well as on the use of more and more industrial development methods, for example:

- The development of the Windows operating system, while seeking to preserve the flexibility that originally existed at Microsoft, introduces more structured aspects, especially the specialization of programmers: specifiers, developers, coders, testers [CUS95][8] ;

- In 1990, in an off-the-shelf software such as Photoshop 1.0.1 [PHO90], there were about 128,000 lines of code ; at the same time in Word 1.1 [WOR91], there were 349,000 lines. A few years later, in 2006, the number of lines of code in Word for Macintosh was approximately *30 million* [MOJ06][9] lines of code (by comparison, the software of the Boeing 787 is made of 14 million lines [LOC13]). Similar comparisons can be done with the most common free software, for example Red Hat Linux 2000 contains around 17 million lines of code [WHE01] (to be compared with the 420,000 lines of embedded software code of the Space Shuttle [NAS96] or either with the 20 million lines of code in Windows NT5 in 1998) ;

It seems that inflation in the size of commercial software is related to factors having no direct relationship with the actual utility that the features have for each user separately, but with the fact that in order for the software to be competitive on the market, it is necessary to package all

7. « Unix and Interlisp, the first integrated programming environments to come into widespread use, are perceived to have improved productivity by integral factors. Why ? They attack the incidental difficulties of using programs together, by providing integrated libraries, unified file formats, and piles and filters. As a result, concepual structures that in principle, could always call, feed and use one another can indeed easily do so in practice. This breakthrough in turn simulated the development of whole toolbenches, since each new tool could be applied to any program by using the standard formats. » [BRO86]

8. « The program manager owns and drives the specification for each release of a product. This person is responsible for soliciting inputs from all groups considered important for the product, especially the developers, who best know the code and what is feasible technically. [...] They do not specify "how" to solve a functional requirement at the level of coding, which developers will do during the design stage. [...] Development and testing groups are responsible for refining the spec during the development process. Developers flesh out details surrounding the functions, estimate the amount of work in person months, and estimate the schedule for their individual pieces of the project. Testers provide early input on whether features seem testable or not, estimate the amount of work in person months for their part of the schedule, and define what they need from development to test and support the product. » [CUS95]

9. « About 30,000,000 lines of code make up the current version of Office that we are developing. That's no typo; in fact, I had to figure that out because of a research project I was working on ... [...] It's really mind blowing when you think about it. Each developer is responsible for, on average, about 428,000 lines of code. [...] Consider for a moment also that there are probably 10x (yes ten times) or more developers on Windows Office, although I don't know the exact count.

[...] there are about 40 lines of text on the page of an average paperback book. That means one developer is responsible for about a 10,700-page book. Or if you break it down smaller, if the average paperback is 300 pages, that means each developer is responsible for about 35 or more paperbacks on his desk. Imagine trying to find a single typo in all those books – that's what most bugs are, don't ya know! » [MOJ06]

the functionalities in one *bundle* [JOE01][10]. Furthermore, this mode of distribution seems to be encouraged by the architecture of the dominant operating systems in existence today, but it is not the only option [KAY12][11].

As soon as the software development techniques become industrialized in the way we succinctly described above, the size of the documentation and the complexity of development environments also explodes, and very highly technical methods are injected in an environment where programmers are largely self-taught [STO15]. For example:

- The Java SE 7 API contains approximately 3,000 classes; assuming that on average each class contains a hundred methods, we deduce that there are approximately 300,000 methods: overall (assuming that a careful reader browsing the documentation spends 10 minutes per method, and that he spends half of his day's work on this activity), this leads us to conclude that, literally, a *lifetime* is not enough to exhaustively read this API[12] ;

- In the Apple environment (using the MacBook Air as an example), the total size (in pages) of the manuals describing the hardware, the programming languages, and the common APIs is around *10,000 pages* [HAG12] ;

- In the Ruby on Rails environment, the toolchain that *programmer apprentices* typically try to learn includes no fewer than eleven languages, APIs and tools, that they believe they must learn simultaneously [GRI12][13] ;

Developing elegant and concise standards (such as those that made the development of the Internet possible, for example) would be a possible approach to improve the situation, but in practice the standards, even when they succeed, turn out to be sometimes exceedingly complex, and are often developed extremely slowly. Moreover, a standard can be implemented in a biased or incomplete way, which can lead to the proliferation of different "improved" implementations of this standard, in a way that allows an individual market player to establish a de facto standard around one

10. « In fact there are lots of great reasons for bloatware. For one, if programmers don't have to worry about how large their code is, *they can ship it sooner*. And that means you get more features, and features make your life better (when you use them) and don't usually hurt (when you don't). [...] A lot of software developers are seduced by the old "80/20" rule. It *seems* to make a lot of sense: 80% of the people use 20% of the features. So you convince yourself that you only need to implement 20% of the features, and you can still sell 80% as many copies.

Unfortunately, *it's never the same 20%*. Everybody uses a *different* set of features. [...] When you start marketing your "lite" product, and you tell people, "hey, it's lite, only 1MB," they tend to be very happy, then they ask you if it has *their* crucial feature, and it doesn't, so they don't buy your product.

Bottom line: if your strategy is "80/20", you're going to have trouble selling software. That's just reality. This strategy is as old as the software industry itself and it just doesn't pay. » [JOE01]

11. « Because what you definitely don't want in a Web browser is any features. [...] You want it to be a mini-operating system, and the people who did the browser mistook it as an application. [...] The job of an operating system is to run arbitrary code safely. It's not there to tell you what kind of code you can run. Most operating systems have way too many features. The nice thing about UNIX when it was first done is not just that there were only 20 system commands, but the kernel was only about 1,000 lines of code. [...] Remember when we had to boot the computer? There's no need for that. There's never been any need for it. Because they did it that way, you wind up with megabytes of features *that are essentially bundled together whether you want them or not*. And now a thousand system calls, where what you really want is objects that are migrating around the net, and when you need a resource, it comes to you — no operating system. » [KAY12]

12. Assuming that our courageous reader doesn't work on weekends, and is entitled to 20 vacation days per year, so she works 360-52×2-20=236 days per year. In a day, if she spends four hours a day reading the API, there are 60×4=240 minutes, so 236×240=56,640 minutes per year available. If we further assume that the Java SE 7 API contains about 300000 methods, and that we spend on average 10 minutes to read each method, we need 3 million minutes to read everything, namely, given the available time, 3,000,000÷56,640=52 years !

13. « Here's the thing: as I said, I pair with novice programmers pretty much every week. And I've come to realize that as a group, *they are faced with a bewildering array of tools to learn*. My typical tutoring student is trying to simultaneously master: Ruby ; Ruby on Rails ; Assorted language mutts and DSLs like YAML, ERB, HAML, and SASS ; Git ; Z shell ; RVM/RBenv ; Homebrew ; Vim. They are trying to learn all of these at once because they've picked up on the consensus opinion that these are all part of the best-in-class Ruby on Rails development toolchain. » [GRI12]

of its products [LEV98][14]. All in all, for various reasons, we're therefore witnessing an extreme complexification of computer technology, which doesn't makes the lives of users any easier.

The evolution (i.e. to make the software hardware-independent) that started during the development of the first operating systems has therefore not yet found its culmination: in practice today, in many cases we no longer depend on the hardware, but instead (in the absence of a standard at the application level), we depend on highly complex environments, which are often developed in a decidedly too patchy and disorganized way.

1.3 Problems that all this poses for software development

Thus, in the situation described above:

- In small projects, it becomes difficult to ensure a good level of quality without substantial expertise (expertise which, as we have seen, is often based on specialized knowledge of products, rather than based on a genuine knowledge of the underlying concepts [STR10][15] ; programmers are losers in that game, since they are thus driven to invest in superficial knowledge [STR10][16], and to develop skills that are gonna lose value quickly) ;

- In larger projects, there is a glass ceiling in quality, because there are too much components which cannot be adapted when the need arises, since they are opaque, and in the hands of teams pursuing their own aims [BOE00][17] ;

All in all, we are only able to implement correctly what the existing components frameworks make easy to achieve ; otherwise, these frameworks *reduce the space of possible designs* on the one hand, and on the other hand, obscure the underlying architectural concepts and make it more difficult to standardize software[18].

Of course, all this encourages the development of poor practice: in particular, the *loss of overall understanding*[19] that arises from the use of misunderstood frameworks makes rigorous testing extremely difficult to implement, for in such a situation, the assumptions about the system become impossible to formulate accurately. Writing good specifications, and implementing the software derived therefrom, may then not be performed satisfactorily. We are finally facing a well-known

14. « All of the Unix vendors wished for *their* brand of Unix to have a proprietary edge over the others. Thus, it was to their advantage to add "features" that didn't exist on other Unices. The goal was to lock developers into one vendor's brand. If they were successful, the resulting software just wouldn't work well on other systems » [LEV98]

15. « For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries (with only the vaguest idea of what's going on). The notions of "maintenance" and "code quality" are typically forgotten or poorly understood. » [STR10]

16. « Industry wants to rely on tried-and-true tools and techniques, but is also addicted to dreams of "silver bullets", "transformative breakthroughs", "killer apps", and so forth. [...] this leads to the development of huge proprietary and mutually incompatible infrastructures [...] platform purveyors want something to lock in developers despite the commonality of basic tools. » [STR10]

17. « Further, the economics of software componentry leave system developers with no choice but to incorporate large commercial-off-the-shelf components into their systems. Unfortunately, developers have no way of knowing what is inside these COTS components, and they have no control over the direction of their evolution. » [BOE00]

18. This was already observed by the developers of the UNIX system: finally the inter-hardware portability proved easier than expected to achieve ; the most difficult point is the inter-API portability: « Most seriously, the operating system interface caused far more trouble for portability than the actual hardware or language differences themselves. Many of the UNIX primitives were impossible to imitate on other operating systems; moreover, some conventions on these other operating systems (for example, strange file formats and record-oriented I/O) were difficult to deal with while retaining compatibility with UNIX. » [JOH78]

19. « Basically, we have learned how to build reasonably reliable systems out of unreliable parts by adding endless layers of runtime checks and massive testing. [...] Often, the many layers of software and the intricate dependencies common in designs *prevent an individual – however competent – from fully understanding a system.* This bodes ill for the future: we do not understand and cannot even measure critical aspects of our systems. » [STR10]

combination of factors, which plays a central role in the failure of software projects [LEH14][20].

Thus, the point (a) suggested by Brooks (i.e., "Harnessing market forces to avoid building components that can be purchased") is much more difficult to implement than it seems a priori. As we have seen, in practice today, the rough approximation of the « ideal » components imagined by Brooks in 1986 takes the form of software frameworks which are too big, and whose opacity goes exactly *against* what is needed to develop good software.

But even when elegant and well thought out components are available, their integration into an architecture that goes beyond the scope for which they were originally designed is a fundamental problem: it's the problem of "architectural mismatch" [BOE00][21]. In this context, we are witnessing a proliferation of hacks, and a duplication of efforts, due to the need to properly re-implement a certain number of key components when their quality is unsatisfactory, or when architectural mismatch makes it impossible to integrate them properly.

Finally, another problem is that there are many components that are badly needed, but that nobody develops [BOE00][22]. For example, the introduction of a standard graphics library for C++ [LAU14] was first proposed in 2014 (that is to say, more than *thirty years* after the appearance of C++ !) and to our knowledge, the standardization process has currently not yet been completed ; in the field of user interfaces and the web, many key developments have not been initiated, many ideas simply have never been explored [MUE02][23] ; in the field of systems administration, many fundamental innovations still have not been disseminated [PIK04][24].

Moreover, a number of key components that *should* have been developed robustly a long time ago (for example, in the world of Java libraries, the Flying Saucer library [FLS04] for generating PDF files from XHTML pages ; or the OpenSSL library [OPE98] which is widely

20. « Interestingly, prior studies have recognized, but not emphasized, the central role of implementation and software testing in causing software project failures. In our cases, these two process areas included the highest number of causes and they had a central role at each failure. [...] Making good requirements was likely very difficult in the case companies. The existing products were complex and required extensive technical knowledge. Additionally, the managers, developers, and testers had somewhat different needs, which meant that extensive practical knowledge was needed while documenting the requirements. In three cases, the participants noticed that the existing product was *so complex that it was likely that nobody knew the system fully*. » [LEH14]

21. « [The AESOP project] assumed that a small number of advanced IT components – a user interface generator, an object-oriented DBMS, and two middleware components – could be integrated quickly and cheaply[...]. The result: a schedule overrun by a factor of four, an effort overrun by a factor of five, and slow, unresponsive system performance. Similar approaches on large-scale systems generally experience even worse results, but often remain undocumented.

Fortunately, the AESOP experience was well analyzed by the SE researchers, who identified *architectural mismatch* as the key success inhibitor. » [BOE00]

22. « The [...] NSF Workshop on a Software Research Program for the 21st Century concluded that [...] software developers need to "address a serious problem [...]: '*desperately needed software is not being developed.*' " » [BOE00]

23. « The foundations for the present form of graphical user interfaces were in the 1960s and 1970s. [...] Since the late 1980s, the pace of innovation on the field of user interfaces for PCs *came to a halt*. [...] The concept of direct manipulation and the paradigm of the desktop metaphor do not scale to the vast amount of items we are managing today. They were right in the beginning – today they are inadequate. Innovation is necessary to regain control of our files.

The situation became even more complicated with the tremendous success of the World Wide Web. The user interface for the Web, that got momentum in the early 1990s, unfortunately lacks a profound approach of user-centered design. *No one felt responsible to start an attempt* comparable with that of Xerox PARC's research for the Star computer or Apple's research for Lisa and Macintosh. As a result the Web interface was never integrated into the desktop environment. » [MUE02]

24. « As a stand-alone system, Unix was pretty good. But when you networked Unix machines together, you got a network of stand-alone systems instead of a seamless, integrated networked system. Instead of one big file system, one user community, one secure setup uniting your network of machines, you had a hodgepodge of workarounds to Unix's fundamental design decision that each machine is self-sufficient.

Nothing's really changed today. The workarounds have become smoother and some of the things we can do with networks of Unix machines are pretty impressive, but when `ssh` is the foundation of your security architecture, you know things aren't working as they should. [...] *We really are using a 1970s era operating system well past its sell-by date*. We get a lot done, and we have fun, but let's face it, the fundamental design of Unix is older than many of the readers of [this interview], while *lots of different, great ideas about computing and networks have been developed in the last 30 years*. » [PIK04]

used to encrypt Internet communications) are in fact developed by teams that have only paltry resources [BRO14][25]. Inevitably, the result is that code quality deteriorates [WHE15][26], producing defects which sometimes turn out to be catastrophic [BBC14][27].

So we are in a situation where *too much* (bad) code is written, while far too little is invested to improve the quality of infrastructure (standards and protocols, components and environments that implement these standards and protocols ; reformulation and *simplification* of all) on which depends the technological ecosystem.

As a result, developments are often poorly managed and poorly understood, and they are undertaken in an environment which is also poorly managed and poorly understood. There's a lot of waste, and software development then becomes much more expensive and risky than it should be[28].

It may seem convenient to promote the use of tools designed to make tasks easier to perform by low-skilled labor, and to use less well educated teams[29]. But this way, not investing enough in the necessary renewal of the technological infrastructure, we find ourselves faced with an ecosystem that becomes increasingly chaotic, complex and difficult to master. On the other hand, because we don't invest enough in training, we are faced with an increasing lack of skills, when in fact, they are needed the most to master the complexity of existing environments, and to design the infrastructures of the future. The points (a) and (d) mentioned by Brooks are then very imperfectly implemented.

1.4 Problems related to maintaining the intellectual ecosystem

In such a context, which is characterized by the complexity of the software ecosystem, as well as by the dissemination of software engineering techniques that require great expertise to be truly mastered, it is difficult to find good programmers [MEY01][30][STR08][31][STR10][32]. A root cause of all this is that in fact, if we carefully observe things, we see that the *conceptual* framework of informatics seems to be largely ignored, both in industry and in universities. In complex industrial projects, organizations often seek to simplify and standardize their processes in order to depend as little as possible on the expertise of developers [STR10][33]. In their turn, universities have difficulties

25. « The important role OpenSSL plays in securing the Internet has never been matched by the financial resources devoted to maintaining it. The open source cryptographic software library secures hundreds of thousands of Web servers and many products sold by multi-billion-dollar companies, but it operates on a shoestring budget. OpenSSL Software Foundation President Steve Marquess wrote [...] that OpenSSL typically receives about $2,000 in donations a year and has just one employee who works full time on the open source code. » [BRO14]

26. « [The] OpenSSL code is extremely complex; it includes multiple levels of indirection and other issues that simply exceeded these tools' abilities to find the vulnerability. [...] there should be a continuous effort to *simplify the code*, because otherwise just adding capabilities will slowly increase the software complexity. The code should be refactored over time to make it simple and clear, not just constantly add new features. [...] The goal should be code that is *obviously right*, as opposed to code that is so complicated that *I can't see any problems*. » [WHE15]

27. « Heartbleed was one of the worst internet flaws ever uncovered. The maintenance of [OpenSSL], which secures around two-thirds of the world's websites, was done by a group of volunteers with very little funding. [...] The industry has been forced to step up after Heartbleed brought chaos to the tech sector. Experts estimate that the Heartbleed bug will cost businesses tens of million of dollars in lost productivity as they update systems with safer versions of OpenSSL. » [BBC14]

28. « Reward systems favor both grandiose corporate schemes and short-term results. The resulting costs are staggering, as are the failure rates for new projects. » [STR10]

29. « Industry wants to [...] be able to operate with minimally skilled and interchangeable developers guided by a few "visionaries" too grand to be bothered by details of code quality. This leads to immense conservatism in the choice of basic tools (such as programming languages and operating systems) and a desire for monocultures (to minimize training and deployment costs). » [STR10]

30. « In fact, talking to managers in industry reveals that they are not just looking for employees – they are looking for excellent developers. This is the really scarce resource. » [MEY01]

31. « Basically every company I visit has a shortage of qualified software developers. I have even had questions from strangers in airplanes: "You're a professor? In software? Have you got any students? Here's my card." » [STR08]

32. « In industry, complaints about the difficulty of finding graduates who understand "systems" and "can architect software" are common and reflect reality. » [STR10]

in teaching the software development activity in a realistic manner [STR10][34][STR08][35], and to uphold the required conceptual depth [KER00][36][MEY01][37][BAR05][38].

Specifically, a first problem is that relatively few university professors develop large applications [DEW08a][39]: therefore, they do not organize the projects that would be needed to teach this to their students [STR10][40], while often, aspects related to software engineering are not even considered as worthy topics from an academic point of view [STR08][41]. A second problem is that to attract students, universities tend to look for simplifying the curriculum and to adopt approaches that are on the one hand less rigorous from a theoretical perspective (less mathematics), and on the other hand less demanding in practical terms (less algorithmics): as a result, the knowledge acquired by students is insufficient, and this way, easily replaceable professionals are being trained [DEW08b][42].

We thus have a situation where there are not enough really good programmers, and plethora of programmers with superficial knowledge. A vicious circle is then created, in which the industry is engaging in practices such as outsourcing ; this makes computer studies less attractive ; to overcome the problem, universities seek to make curricula even more approachable, etc. [DEW08a][43].

33. « Let me point to the issue of scale. Many industrial systems consist of millions of lines of code [...] realizing this, many organizations focus on simplifying tools, techniques, languages, and operating procedures to minimize the reliance on developer skills. » [STR10]

34. « A student can graduate with honors from top CS programs without ever writing a program larger than 1,000 lines. » [STR10]

35. « Take a simple example: A friend of mine looked at the final projects of a class of third-year CS students from a famous university. Essentially all had their code littered with "magic constants". They had never been taught that was bad style – in fact *they had never been taught about programming style* because the department "taught computer science; not programming". » [STR08]

36. « Teaching students skills so that they can step immediately into a Windows development shop and are able to write COM programs is just not right. That's not what universities should be doing; universities should be teaching things which are likely to last, for a lifetime if you're lucky, but at least 5 or 10 or 20 years, and that means *principles and ideas*. At the same time, they should be illustrating them with the best possible examples taken from current practice. » [KER00]

37. « In any engineering discipline, [the tools of the trade] constitute a good part of the professional's bag of tricks. We must, however, look beyond them to the fundamental concepts that have not changed significantly since they emerged when the field took shape some 30 years ago. As in hardware design, *the technology evolves, but the concepts remain*. » [MEY01]

38. « A project leader at an aerospace company once told me that in his experience it is easier to teach computing to a physics major than it is to teach physics to a computer science major. Given the firm theoretical foundation of students of even a concrete subject like mechanical engineering compared with *the artifact-laden* education of a computer science student, this statement is not at all surprising. » [BAR05]

39. « Part of the trouble with universities is that there is relatively few faculty members who know much about software. They know about the theory of computer science and the theory of programming languages. But there are relatively few faculty who really are programmers and software engineers and understand what's involved in writing big applications. » [DEW08a]

40. « To integrate that knowledge and to get an idea of how to handle larger problems, *every student must complete several group projects* (you could call that basic software engineering). It is essential that there is a balance between the theoretical and the practical—CS is not just principles and theorems, and it is not just hacking code. [...] Many professors will object: "I don't have the time to program!" However, I think that professors who teach students who want to become software professionals will have to make time and their institutions must find ways to reward them for programming. The ultimate goal of CS is to help produce better systems. Would you trust someone who had not seen a patient for years to teach surgery? What would you think of a piano teacher who never touched the keyboard? » [STR10]

41. « That is, programming was seen as a lowly skill that students either did not need or could easily pick up on their own. I have seen the result of that attitude in new graduate students: It is rare that anyone thinks about the structure of their code or the implications for scaling and maintenance – *those are not academic subjects*. » [STR08]

42. « Over the last few years we have noticed worrisome trends in CS education. [...]:

1. Mathematics requirements in CS programs are shrinking.

2. The development of programming skills in several languages is giving way to cookbook approaches using large libraries and special-purpose packages.

3. The resulting set of skills is insufficient for today's software industry (in particular for safety and security purposes) and, unfortunately, matches well what the outsourcing industry can offer. *We are training easily replaceable professionals.* » [DEW08b]

43. « The situation has the potential to turn into a vicious circle: Enrollment is down, so CS programs lower requirements. This in turn creates more CS grads who are replaced by outsourcing. As a result, news of this

Moreover, the industry wants excellent developers, but even when they can be found, companies don't often seem eager to give them the freedom they need to express themselves [STR08][44]. Overall, a short-sighted and superficial attitude remains widespread [STR14][45], whose mediocrity probably discouraged more than one calling [STR10][46].

Indeed, the computer industry has long used utterly toxic practices, for example by taking advantage of discriminatory legislations which, in various industrialized countries, enable companies to hire foreign employees at prices lower than market value (e.g. in the USA, holders of H1-B visas, which are de facto placed in a precarious situation that has been compared to a kind of « indentured servitude » [MAT03][47]. In Europe, similar practices exist, we do have this for example in France [MUN14][48]). Another example is the anticompetitive agreement that a number of leading companies of the Silicon Valley have concluded in order to prevent their employees to capitalize on competition for access to better career opportunities [REU14][49].

Arrived here, we can only be struck by the very clearly expressed preference of many key players in the industry for what might be called *immobile* labor [MAT03][50], that is to say, so that one way or another, may be created situations where employees are effectively deprived of part of their freedom.

Under such conditions, the question can be asked: how do the leaders of that industry envision *innovation* ? The author of this article strongly doubts that such a morbid focus on a (pseudo) loyalty of their employees, which apparently, the employer and its agents only imagine as compelled and forced, could be conducive to creativity, or even to innovation.

---

outsourcing keeps driving down CS enrollment – and the spiral continues. » [DEW08a]

44. « The companies are complaining [...]. They correctly see a shortage of good developers as a part of the problem. What they generally don't see is that inserting a good developer into a culture designed to constrain semi-skilled programmers from doing harm is pointless because the rules/culture will constrain the new developer from doing anything significantly new and better. » [STR08]

45. « People consistently underestimate the intellectual effort needed to design, implement, and maintain software. Many consider it – and especially programming – a low-level, "manual" skill. [...] The current fashion of more focussed, more efficient, and cheaper education exacerbates this trend. Good system design and implementation is a highly skilled job deserving the respect granted to the highest levels of professions. However, the current educational systems and the almost complete lack of professional-level life-long education do not produce such professionals in high numbers and *they often get drowned in a mass of clever, hardworking, but miseducated programmers* caring less about the longer term of the systems they build. Many managers of IT projects are equally short-sighted. » [STR14]

46. « The idea of software development as an assembly line manned by semi-skilled interchangeable workers is fundamentally flawed and wasteful. It pushes the most competent people out of the field and discourages students from entering it. » [STR10]

47. « the H-1B workers are *de facto* indentured servants; this, together with gaping loopholes in H-1B regulations, enables employers to attain Type I labor cost savings.

In addition to the issue of cheap labor, some employers euphemistically refer to the "remarkable loyalty" of the H-1B workers. Since an H-1B is typically in no position to seek other employment, the employer need not worry that the worker will suddenly leave the employer in the middle of a pressing project. In addition, the employer can force the H-1B to work long hours. To many employers, this "loyalty" aspect is the prime motivation for hiring H-1Bs, whether or not they are saving salary costs in doing so. » [MAT03]

48. « It seems that very often in France, these foreign computer scientists earn *wages below those of their French counterparts* (on average, 15-20 percent less) [...] to retain their work permit, foreign workers avoid changing employers during the first years so as not to expose themselves to the frequent risk of a non-renewal of their permit by the prefectures (*"change of employer not justified"*). » [MUN14]

49. « Tech workers filed a class action lawsuit against Apple Inc, Google Inc, Intel Inc and Adobe Systems Inc in 2011, alleging they conspired to refrain from soliciting one another's employees in order to avert a salary war. Trial had been scheduled to begin at the end of May on behalf of roughly 64,000 workers in the class. [...] Google's human resources director [asked] Schmidt about sharing its no-cold call agreements with competitors. Schmidt, now the company's executive chairman, advised discretion. [...] [Steve] Jobs threatened Palm with a patent lawsuit if Palm didn't agree to stop soliciting Apple employees. However, then Palm Chief Executive Edward Colligan told Jobs that the plan was "likely illegal," and that Palm was not "intimidated" by the threat. » [REU14]

50. « Most H-1Bs hope to be sponsored by their employers for permanent residence, i.e. green cards. This is a multi-year process. [...] During the time an H-1B's green card application is being processed, he/she is essentially *immobile*; switching employers during this time would necessitate starting the green card process all over again, an unthinkable prospect for most. » [MAT03]

Indeed, legislation (and how some organizations use it) may, in some cases, act as a barrier to innovation [HIP15a][51]: in practice, innovation is inhibited when firms seek to exercise excessive *control* over the actions that individuals may wish to carry out. It is then essential to keep in mind that in reality, the law provides a robust *right to innovate* to individuals [HIP15a][52]. From there, seeking the most appropriate means to protect their rights and to pursue their innovative activities is up to them.

For example, *user innovation by individual citizens* is a new and important phenomenon that has developed over the last 30 years [HIP15b][53]. In particular, for this form of innovation to be working well, producers and users should seek *complementary* roles, and investments (particularly in the field of education) to increase the number of innovative users seem required, that often have the character of a public good [HIP15b][54] (which also means that this kind of investment is not typically undertaken by producers).

For the intellectual ecosystem of informatics to be preserved, the computer industry needs better developers [STR10][55], and in universities, the curriculum must fundamentally evolve to provide solid theoretical foundations for students [BAR05][56].

But the industry must also evolve: as evidenced by, for example, the massive use of necessarily superficial keywords [STR08][57], recruiting organizations are often unable to assess the actual abilities of the candidates they encounter, and are also unable to give them the freedom without which real innovation is inconceivable. One day or another, il will be necessary to finish with these kind of attitudes: otherwise improving the education of developers will have no prospect for helping companies to solve their problems, which come primarily from an underestimation of the role played by *knowledge* in the functioning of innovation, as well as in the functioning of industry itself. An alternative might be the emergence of a more open software industry, which would know how to work in harmony with users, and would truly know how to encourage them to innovate.

51. « However, we also show that legislation and regulation—often promulgated without awareness of consumer innovation as a valuable resource—can, in practice, significantly interfere with individuals' exercise of their fundamental freedom to innovate. This interference can cost society dearly by discouraging and slowing innovation or even thwarting it entirely. » [HIP15a]

52. « Fortunately, the law already provides a robust right to innovate to individual or collaborating inventors, designers, creators, and tinkerers who inhabit the innovation wetlands. The core of this right protects noncommercial innovation for personal use, collaboration with other similar innovators, and free dissemination to others of information about innovations that result from these activities. » [HIP15a]

53. « Innovation has traditionally been seen as the province of producers who invest in product and service development in order to sell their innovations. However, extensive theoretical and empirical research has now led to an understanding that *users* are also a major source of innovation development, where users are defined as entities that develop novel products and services for use rather than sale.

Over the last three decades, user innovation by individual citizens has moved from being considered an anomaly to being recognized as an activity conducted by many millions of users that results in the creation of many individually and commercially important new products and services. » [HIP15b]

54. « Our findings suggest that policy measures directed at elevating the share of innovating users [...] will increase welfare even further. [...] In essence, in a world with synergistic investments of firms and users, you need both innovating users and non-innovating users who benefit from their efforts to maximize welfare. Although everyone, including producers, would gain from measures that turn non-innovating users into innovating users [...], producers may nonetheless not undertake such measures on their own. The types of investments required often have the character of investment in a public good, which typically results in private under-investment. » [HIP15b]

55. « For a viable alternative, industry needs better developers. [...] academia must produce more graduates with relevant skills and industry must adopt tools, techniques, and processes to utilize those skills. » [STR10]

56. « I believe that CS education must fundamentally change in order to equip the student with a firm, deep and broad theoretical background, long before specialization is undertaken. » [BAR05]

57. « Too often [companies] search for people based on a narrow model based on currently fashionable buzzwords. In doing so, they miss many experienced, well-educated developers. » [STR08]

2 Considerations about the nature of knowledge

Exactly for the sake of what is new and revolutionary in every child, education must be conservative; it must preserve this newness and introduce it as a new thing into an old world, which, however revolutionary its actions may be, is always, from the standpoint of the next generation, superannuated and close to destruction.

Hannah Arendt – The crisis in Education

I learned a lot from my students at the Ecole Normale. Many of them prepared theses under my direction – one normally says "direction", but my "direction" consisted in understanding what they had in mind. So I learned very much. [...] The students had to be helped of course, but they had their own ideas. Each one has his own personality, and you have to respect this personality, to help him find his own personality, and certainly not to impose somebody else's ideas.

Henri Cartan – interview, Notices of the AMS, 1999.

2.1 History of ideas and dissemination of knowledge

2400 years ago died Socrates, sentenced to death for defending an uncompromising vision of truth[58], and for having criticized the hypocrisies of the society of his time [PLATOa][59]. As the Chinese philosopher Confucius before him [XIA07], Socrates makes use of an heuristic method which is open to all, and aims to enable the learner to rediscover *inside herself* the knowledge that some profess to teach [PLATOb][60]. At that time in Greece, and later in Rome, there used to be professionals who taught rhetorics, an essential skill in these societies where social success was often attained by mastering the art of knowing how to speak and convince. We see reflected here all the ambiguity then pertaining to the use of speech, which can be used to organize collective action in a manner consistent with the public interest, or rather to serve vested interests.

In the Middle Ages in Europe, the governance of societies having evolved in a frankly aristocratic direction, the formal mastery of oratory skills, literature and science become the privilege of the nobility and clergy, a phenomenon amplified by the fact that vernacular languages having also evolved, the *language* itself in which books are written (i.e., Greek and Latin) had then become inaccessible to most. At that time, education is of a dogmatic and magisterial nature, and it is based on religious principles and on the abstract concepts of philosophy and scholasticism. The development of science and technology progresses continuously, however, but exists a clear divide between the academic expertise of the clergy and the technical expertise of artisans, who are often talented engineers (the cathedrals, or either the increasingly sophisticated weapons that are developed during all the Middle Ages, have obviously not been designed by amateurs).

58. « The unexamined life is not worth living [...]. I would rather die having spoken after my manner, than speak in your manner and live [...]. The difficulty, my friends, is not to avoid death, but to avoid unrighteousness » [PLATOa]

59. Especially, Socrates abhors the purely utilitarian vision that his countrymen have of power (i.e., « [You] are [...] from the city that is greatest and best reputed for wisdom and strength: are you not ashamed that you care for having as much money as possible, and reputation, and honor, but that you neither care for nor give thought to prudence, and truth [...] ? »), and he blames them for attaching importance to knowledge only when it seems able to provide material advantages, that is to say: « [to] regard the things worth the most as the least important, and the paltrier things as more important. » [PLATOa].

60. « [Education] is not what the professions of certain men assert it to be. They presumably assert that they put into the soul knowledge that isn't in it, as though they were putting sight into blind eyes. [...] But the present argument, on the other hand indicates that this power is in the soul of each, and that the instrument with which each learns [...] must be turned around from that which is *coming into being* together with the whole soul until it is able to endure looking at that which *is* and the brightest part of that which *is*. » [PLATOb].

The Renaissance is the period when the contradictions accumulated during the previous period lead to a breakthrough, which first sees the stranglehold of the Catholic Church on spiritual matters be challenged by the Protestants, who in particular arrange for the translation of the Bible into the vernacular, and thus reclaim the language in which it is written. The development of the experimental method in science (e.g. Bacon) then inescapably leads to the reconsideration of the literal reading of religious texts that was practiced until then (Galileo's trial is but one famous episode of that transformation), and, in turn, stimulates a movement of democratizing knowledge that gradually comes to assert itself explicitly [DES29][61][ERA11][62]. Finally, the invention of printing by Gutenberg creates the technical means of disseminating knowledge on a large scale, and turns a questioning which initially, was the deed of a minority, to an irreversible evolution.

In the modern period, there is a radical challenge being issued to the traditional order of European societies: the movement of political democratization that began in England develops further in France, and the political regimes throughout Europe finally move towards the parliamentary form of government that they have today. An important consequence of these developments is the rise of industrialization and the development of mass education, which is the source of the dynamism of modern industrial societies and of their perpetual regeneration.

Arrived here, we see that the evolution of ideas seems to translate into a pendulum swing between on the one hand a number of attitudes and pre-existing elements which, alive and well initially, turn to a dogmatism which little by little settles, fossilizing thinking into overly complex forms (e.g., those of the sophist philosophies of Antiquity, of the scholasticism of the Middle Ages) which empty it from its original creative power; and secondly a reappropriation of the tools of thought (of the logic by Socrates, of the language by the Protestants, of thought itself by the humanists and by the philosophers of the Enlightenment) which occurs sooner or later, and then shatters the crumbling framework of the established tradition.

2.2 Language as a sociocultural fact

Over the past 50 years, a number of research have been developed, which shed a new light on the question of the nature of thought as an individual and collective phenomenon in human societies.

Firstly, it seems that to be fully understood, cognition should be regarded not as a phenomenon that can be explained by the sole study of knowledge as writing, but that rather, it can not be separated from the *social* forms through which cognitive activity is actually always materialized: human cognition presents itself as a *situated activity* which, first, helps to maintain the cohesion of human communities and of the changing roles that individuals play in these communities, and contributes to the evolution and development of their personalities (dimension which includes, but is not limited, to learning-related aspects as such) [HEN04][63]. Learning, and in general, many of the interactions that can occur in a community, make use of the environment as a *medium*, which acts as an external memory, which individuals use to encode and decode the representations by means of which communication can then, to a large extent, function *implicitly* [HEN04][64].

61. « Anyone who has learned this whole method perfectly, however humble his abilities may be, will nevertheless perceive that *none of these ways is less open to him than to anyone else*, and there is nothing further of which he is ignorant because of any failure of ability or method. » [DES29, Eighth rule]

62. « And herein they copy after the example of some rhetoricians of our times, who think themselves in a manner gods if like horse leeches, with a double-tongued fluency they can plead indifferently for either side, and deem it a very doughty exploit if they can but interlard a Latin sentence with some Greek word, though altogether by head and shoulders and less to the purpose. And if they want hard words, they run over some worm-eaten manuscript and pick out half a dozen of old obsolete terms to confound their reader, believing, no doubt, that they that understand their meaning will like it the better, and they that do not will admire it the more by how much the less they understand it: my friends love it from afar. » [ERA11]

63. « The unstated normative view of learning for most of us is derived from our school experience. The view of learning often is that it is somewhat like medicine—it is not supposed to taste good, but it will make you better, or in the case of learning in school, remedy an inherent defect that the student has when he or she enters the class. From this point of view, learning is supplied (delivered) to the learners rather than being demand driven by learners. [...] It is not enough for schools to justify what is to be learned by claiming that it is relevant to some real world activity. Learning becomes demand driven when the need to learn arises from the desire to *forge a new identity* that is seen as valuable. » [HEN04]

Secondly, it seems that communication originates in *gestures*, which serve to point, that are found in great apes [TOM08][65] ; one might think that the situated nature of human activities naturally stems from there. In evolutionary terms, the ability to think beyond the limits of the here and now (the development of which is programmed in children [PIA69]), which requires the ability to *represent* things accurately, is linked to the development of the ability to encode the description, thus to explicitly conceptualizing and *communicating* it, therefore to the emergence of a language having a complex grammatical structure, that is not found in great apes. In particular, the difference in nature between great apes and humans can not be solely explained by a difference in terms of "general intelligence", the essential difference is the qualitatively superior development of socio-cognitive skills in humans [HER07][66]. Different hypotheses have been put forward: it can be assumed that during the process of hominization, socio-cognitive skills developed first (which subsequently led to the development of the conceptually abstract language of humans [TOM08][67]). But it is also possible that the development of the ability to perceive increasingly complex causalities in the physical world appeared first, and finally made humans capable of explicitly conceptualizing the underlying inner workings of their peers' behavior [HER07][68].

Anyway, the use of a complex conceptual language is an important evolutionary advantage, since it enables the individuals to develop *plans* through which sophisticated collective actions can be conducted. From there, the information provided by an individual is no longer limited to simple *requests* that are solely intended to influence the interlocutor's behavior in the here and now [TOM08][69], but rather, it comes in the form of *stories* describing complex hypothetical scenarios set in a more or less near future. Here appears an explicit conceptualization of the *role* that individuals can play in the plan which has been communicated, which illuminates a little more the socially situated nature of human activity that we described above: particularly, we see how from the structure of the language and the activities it makes possible, is being built the *identity* of individuals in a group. This had already been noticed by Vygotsky: « Who we become depends on the company we keep and what we do and say together » [WEL09]. For Vygotsky, the development of specifically human cognitive abilities can only take place through interaction

64. « The creation of a valued social identity shapes learning and provides the interpretive resources that are embedded in a particular community of practice. These interpretive resources are used to make sense of the representations that are constructed in language, bodily posture, and artifacts by members of the community for public display. The local appropriation of the meaning of these representational displays in turn contributes to the construction of competent knowledge in use which furthers the formation of desired identities. [...] The representations that have been of interest in this chapter are produced in such a way that they are made visible to members of a community of practice (an interacting group) *without the need for overt explication* by the members of the group. [...] The nuanced changes in these representations appear to an outside observer as nonsensical or trivial to the task at hand, yet for the members these changes [...] are fundamental to the creation of an ongoing sense of what is actually happening. [...] When learning is seen from a participation metaphor [...] the movement into full participation *depends fundamentally* on being able to read the representations that are socially produced for common display. » [HEN04]

65. « The road to human cooperative communication begins with great ape intentional communication, especially as manifest in gestures. » [TOM08]

66. « Supporting the cultural intelligence hypothesis and contradicting the hypothesis that humans simply have more "general intelligence," we found that the children and chimpanzees had very similar cognitive skills for dealing with the physical world but that the children had more sophisticated cognitive skills than either of the ape species for dealing with the social world. » [HER07]

67. « With the emergence of the informing function and referents displaced in time and space, there arises a need for grammatical devices to (i) identify absent referents by grounding them in the current joint attentional frame (perhaps using multiunit constituents), (ii) syntactically mark the roles of participants, and (iii) distinguish requestive from informative communicative motives. » [TOM08]

68. « However, we should note that because the children were somewhat more skillful than the apes in the causality tasks not involving active tool manipulation, as well as in the tasks of social cognition, it is possible that what is distinctively human is not social-cultural cognition as a specialized domain, as we have hypothesized. Rather, what may be distinctive is the ability to understand unobserved causal forces in general, including (as a special case) the mental states of others as causes of behavior. » [HER07]

69. « Apes always use their learned, intentional gestures to request/demand actions from others, including humans. They use their intention-movements to demand action directly. They use their attention-getters to demand action indirectly, that is, they use them to direct the other's attention so that she will see something and then do something as a result. [...] When "linguistic" apes—and so perhaps very early humans—produce multiunit utterances, they use them almost always for requestive functions—which typically involve only "me and you in the here and now," [...]. These apes and early humans thus have only a grammar of requesting. » [TOM08]

with others, and (once acquired) higher mental functions can be considered as internalized social relationships [VYG81][70].

In addition to making possible formidably effective forms of cooperation, the existence of the language that underlies them probably exerts a selection pressure favoring the "linguistic" individuals, because there are experiences to which one has not been exposed directly: acquiring knowledge about these experiences boosts the evolutionary advantage of the communities to which such individuals belong. This is perhaps why, despite their unreliability, a great importance is generally given to *testimonies* in human societies [DAV12][71].

Finally, in addition to being the medium through which knowledge can be encoded in the form of stories, language *itself* works as a long-term accumulator of knowledge, since to a large extent, it is built using metaphors (this is what could be termed the "folk wisdom" of language) [LAK80][72].

2.3 Importance of knowledge in pedagogy

Arrived here (humans being on the one hand programmed to *implicitly* learn during their development [PIA69], and secondly, to *implicitly* grasp the meaning of the representations produced by themselves and others throughout the socially situated interactions that they lead in their lives [HEN04]), there seems to be good reason to consider that, in general, learning is socially situated, and that a pedagogical approach which does not separate knowledge from its social instantiation has good chances to be promising.

Hence the idea that the ability to grasp the implicit signals that can be found in the environment is fundamental to learning, and that education should be based on discovery, that it should allow students to rediscover knowledge by themselves [PIA73][73] from experiences that are meaningful for themselves [DEW38].

But in practice, the constructivist pedagogies inspired by this kind of approach seem rather difficult to validate by means of controlled experiments: on the contrary, experience shows that guided instruction is much more effective than unguided instruction [KIR06][74]. In particular, numerous evidence shows that people learn better by working from *examples*, rather than by trying to solve problems without information to guide the resolution (this is what is called the « worked example effect ») [SWE85]. This suggests that the learning of a number of school and academic knowledge (even in areas apparently as basic as learning to read, for example [EHR01]) requires a *conscious* effort, and therefore does not work in the implicit manner that we would have expected.

70. These specifically human cognitive abilities are « a copy from social interaction; all higher mental functions are internalized social relationships » [VYG81].

71. « The reliability claim loses its plausibility in the light of the levels of interpretive filtering required by testimony's nature as a non-basic source, if not straightforwardly as the result of psychological studies on its reliability. There seems, however, to be no empirical or conceptual objection to the prima facie plausibility of the scope claim as we have presented it. Given the domain generality of testimony as compared with perceptual belief sources, a good case may be made for its greater scope, or representational power, relative to them. » [DAV12]

72. « Primarily on the basis of linguistic evidence, we have found that most of our ordinary conceptual system is metaphorical in nature. [...] To give some idea of what it could mean for a concept to be metaphorical and for such a concept to structure an everyday activity, let us start with the concept ARGUMENT and the conceptual metaphor ARGUMENT IS WAR. [...] It is important to see that we don't just *talk* about arguments in terms of war. We can actually *win* or *lose* arguments. We see the person we are arguing with as an *opponent*. We *attack* his positions and we *defend* our own. We *gain* and *lose ground*. We *plan* and use *strategies*. If we find a position *indefensible*, we can *abandon* it and take a new *line of attack*. Many of the things we *do* in arguing are partially structured by the concept of war. [...] metaphor is not just a matter of language, that is, of mere words. We shall argue that, on the contrary, human *thought processes* are largely metaphorical. » [LAK80]

73. « To understand is to discover, or reconstruct by rediscovery, and such conditions must be complied with if in the future individuals are to be formed who are capable of production and creativity and not simply repetition. » [PIA73]

74. « None of the preceding arguments and theorizing would be important if there was a clear body of research using controlled experiments indicating that unguided or minimally guided instruction was more effective than guided instruction. In fact, [...] the reverse is true. Controlled experiments almost uniformly indicate that when dealing with novel information, learners should be explicitly shown what to do and how to do it. » [KIR06]

Here comes in the fact that despite humans are biologically programmed to acquire a large number of essential skills for interpretating the world (in particular, language, and concepts necessary for a fine intuitive understanding of the physical world, of the biological world, of psychology), the technical and cultural *knowledge* that have been built over generations are much newer in regard to the ladder of evolution. And as they develop, these knowledge acquire an increasingly complex structure, which corresponds less and less to the intuitive way of understanding the world that comes from our biology. They are of an artificial nature, and are "coded" in a conceptual system which is also artificial (and which moreover, is implicitly perceived by insiders, but remains opaque for the uninitiated). This is why these structured artificial knowledge of technical and cultural nature, which are the product of works and discoveries accumulated over generations, are not likely to be easily rediscovered by a single individual: they must be *explained*. In practice, there is a very clear difference between the naïve interpretation that people have of a given domain (e.g., physics) and the rigorously verified knowledge that we have on this same domain. In other words, there is a gap between the "folk" understanding of a domain and the understanding that educated persons have of this domain [GEA07].

Folk psychology is indeed of great interest for pedagogy, since it allows experts (who have often forgotten much of the intellectual journey they have accomplished in their own learning) to get a better idea of how other people grasp the domain, as well as of the essential steps in the progression that leads to the intellectual mastery of the domain (the "threshold concepts" [MEY03][75]). Another essential aspect is that the educated persons' way of thinking can *also* exhibit elements pertaining to folk psychology, resulting from the fact that from inside their own expert vision, they can be unaware of their own prejudices and *implicit* attitudes: the awareness that leads to the explicitation of this is also a "threshold concept", which leads to a deeper understanding of the domain [MEY03][76].

Arrived here, we note that all cultural knowledge therefore prove to be of an artificial, very technical nature, which as we have said, must be acquired by means of a conscious learning. But on the other side (like the example just cited shows), once assimilated, they then work in the *implicit* and *socially situated* manner which is naturally going on in human communities, with the result that practice in a domain (and therefore inevitably, the way it is taught) will always be colored with strongly implicit aspects, *putting at a disadvantage* those who have not had the chance to live in an environment which made the implicit innuendos that are used in teaching easy to grasp for them. This is why education must be based on solid explicit methods to make the implicit undertones which are inevitably needed clear *to everyone* [BOU66][77]. In this sense, education must be of a *conservative* nature: because even if in the short-term, it may seem easier and therefore less unequal, any kind of educational innovation that fails to transmit the tradition that stems from the great works of the past will always in the long run end up *harming* the individuals who do not have access to the complex cultural capital that one has seen fit to avoid making the effort to

75. « A threshold concept can be considered as akin to a portal, opening up a new and previously inaccessible way of thinking about something. It represents a transformed way of understanding, or interpreting, or viewing something without which the learner cannot progress. As a consequence of comprehending a threshold concept there may thus be a transformed internal view of subject matter, subject landscape, or even world view. This transformation may be sudden or it may be protracted over a considerable period of time, with the transition to understanding proving troublesome. Such a transformed view or landscape may represent how people 'think' in a particular discipline, or how they perceive, apprehend, or experience particular phenomena within that discipline (or more generally). » [MEY03]

76. « As the study of music becomes increasingly multicultural, possible clues as to the existence of other tuning systems are sometimes encountered, but the tendency to Westernise such cultures in terms of popular music once again asserts the dominance of equal temperament. The chance hearing, perhaps, of an Indonesian gamelan orchestra may lead a student to observe that the gongs appear to be 'out of tune', but it is rare indeed that they recognise the significance of alternative tuning systems in the development of other musical genres in Asia and beyond. Thus it is that an understanding of tuning methodologies and their evolution through history and across the world becomes a threshold concept for an advanced understanding of pitch organisation in music. » [MEY03]

77. « By failing to provide to all, through a systematic education, what some owe to their family environment, the school system therefore endorses inequalities that it alone could reduce. Indeed, only an institution whose specific function is to transmit to the greatest possible number, by learning and exercise, the attitudes and skills that constitute a cultivated man, could compensate (at least partially) the disadvantages of those who do not find the incentive to cultural practice in their family environment. » [BOU66]

teach them. Thus, we will have educated a generation of adults who may be collectively incapable of solving the problems they have to face [ARE58][78]. The right approach, therefore, is to gather first *the best of the existing tradition*, and to try to clarify it in the best possible manner to give ourselves the means to transmit it to all.

Finally, note that experts' cognition is based on the exploitation of a *large amount* of knowledge: in chess, for example, the ability to recognize significant patterns plays a fundamental role in the expertise [DEG65][79] ; it is because they know *a lot of examples* that experts are able to recognize situations, and to classify problems according to the major principles governing their resolution rather than according to surface features of the problem [CHI81], and thus make better choices during their problem-solving activities. Hence the importance of providing a large number of *good* examples to facilitate learning [SWE85]. Finally, the *scientific method* itself is based on the iterative development of models, an activity that often requires collecting a large number of observations, along with knowledge of a large number of theoretical elements. Learning *modeling* requires a lot of practice in order to acquire a sufficient number of good quality examples [LEH00][80]. In many other areas, one can make similar observations [BLO85][HAY89].

Therefore, we are led to realise that many of the know-hows on which are built the modern industrial societies in which we live are of an artificial, highly constructed nature, and what's more, of a very *knowledge intensive* nature. This is why their acquisition requires *long* periods of learning. Nevertheless, the acquisition of expertise can not be reduced to a purely instrumental vision which considers that, roughly, learning consists of the acquisition of a large amount of knowledge that somehow we could just inject into the minds of learners. The problem is more complicated than that, because as well as being numerous, the knowledge to be assimilated also turns out to be of great diversity: there is lots of knowledge, but in addition, there are also many *types* of knowledge and skills, whose interactions need to be understood. In the case of learning to read, for example, it appears that an instrumental knowledge of the code is *not enough* to make individuals able to handle writing in a truly meaningful way [WEL09][81]: in fact, the key skill, of a much more fundamental nature that seems to be related to this, is the ability to think *beyond the limits of the here and now* [WEL09][82], whose fundamental importance in the functioning of human cognition has

78. « To avoid misunderstanding: it seems to me that conservatism, in the sense of conservation, is of the essence of the educational activity, whose task is always to cherish and protect something: the child against the world, the world against the child, the new against the old, the old against the new. Even the comprehensive responsibility for the world that is thereby assumed implies, of course, a conservative attitude. But this holds good *only for the realm of education*, [...] and not for the realm of politics, where we act among and with adults and equals. In politics this conservative attitude–which accepts the world as it is, striving only to preserve the status quo–can only lead to destruction, because the world, in gross and in detail, is irrevocably delivered up to the ruin of time unless human beings are *determined to intervene*, to alter, to *create what is new*. » [ARE58]

79. « We know that increasing experience and knowledge in a specific field (chess, for instance) has the effect that things (properties, etc.) which, at earlier stages, had to be abstracted, or even inferred are apt to be immediately perceived at later stages. To a rather large extent, *abstraction is replaced by perception*, but we do not know much about how this works, nor where the borderline lies. » [DEG65]

80. « Modeling must be practiced systematically so that the forms and uses of a variety of models are explored and evaluated. Lesh (personal communication, June, 1995) suggested that being a good modeler is in large part a matter of having a number of fruitful models in your "hip pocket." Acquiring such a collection almost certainly requires sustained work in a context where modeling has a purpose and a payoff. » [LEH00]

81. « Is [how much a child knows about the conventions of written language] what explains the difference between those who quickly and easily learn how to read and write and those who do not? Or is it more a byproduct of acquiring a different sort of knowledge about literacy that is of much greater fundamental importance? [...] The point is perhaps made more clearly if we draw the comparison with spoken language. What use would it be for a child to know all the sounds of his or her language and the rules for combining them into words and sentences, if the child had not discovered that the whole point of this knowledge was that it was a resource for the exchange of meanings in the attainment of joint purposes of various kinds? The same is true, I believe, for an understanding of the mechanics of literacy. Important though this knowledge is for reading and writing, it is of little value if the child does not understand the purpose of these activities – if he or she does not know that written language conveys meaning and that it does so in ways that differ from those that apply in the use of spoken language. » [WEL09, p. 167]

82. « through stories, children vicariously extend the range of their experience *far beyond the limits of their immediate surroundings*. In the process, they develop a *much richer mental model of the world* and a vocabulary with which to talk about it. As a result, as the content of the curriculum expands beyond what can be experienced firsthand in the classroom, children who have been read to find themselves *at a considerable advantage*. » [WEL09, p. 171]

already been mentioned. And the study described in [WEL09] clearly shows that the key activity that promotes the acquisition of the ability to think beyond the limits of the here and now is *reading stories* [WEL09][83] (furthermore, the importance of reading stories was also noted by [GEA07][84]).

Arrived here, it appears that a very effective way of scaffolding learning consists of knowing how to *identify the context* in which what we wish to teach takes its true meaning, and then to place ourselves in that context: thus, knowledge and skills that belong to this context complement and support learning. In the case of reading that we mentioned above, it is possible, and even necessary, to use the activity of reading stories to scaffold learning. Can we use such strategies in other areas ? That question warrants consideration, and it seems that whenever possible, teaching methods should seek to reflect this *culturally situated* aspect of knowledge, which alone has the capacity to make it fully intelligible.

2.4 Importance of motivation and freedom for the development of the mind; criteria for creating an education based on discovery

Understanding starts with a question; not any question, but a real question. Said in another way, a real question expresses a desire to understand. This desire is what moves the questioner to pursue the question until an answer has been made. Desiring to understand opens ourselves to experiencing what is new as new, and the already known under new aspects.

> A. Bettencourt, *On what it means to understand science* (1991), Michigan State University (unpublished paper). Cited in [WEL00]

In spite of appearances, creativity in a human being is an inseparable attribute of its soul, and it is as indestructible as the soul itself.

Whether you discover yourself or mathematics, without desire, any so-called work is but a pretense, which leads to nowhere. [...] It is on this very same night, it seems to me, that I understood that desire *to know and* power *to know are one in the same thing. As soon as we trust and follow it, it is desire which leads us to the heart of those things we long to know. And it is desire, too, which makes us find, without even searching for it, the best method to know these things, and which is the most suitable to our being.*

> Alexandre Grothendieck

Finally, should we abandon the ambition of Socrates, who taught that « I know only one thing, and that is that I know nothing » ? We would then back away from the very *attitude* from which the modern world was built ; for example, Descartes's philosophy actually begins with <u>*doubt*</u> (Descartes teaches that in fact, when one really reflects on it, one can only be struck by the potential lack of

83. « How then might some of the children in our study have discovered something of the significance of written language before they came to school? To answer this question it is necessary to look more carefully at the sorts of activities that they engaged in, or observed, during the preschool years at home [...] the next stage of the investigation [...] involved a careful scrutiny of all the transcripts of the actual observations. [...] The results of this comparison were absolutely clear-cut. Of the three frequently occurring activities that had been considered as possibly helpful preparation for the acquisition of literacy, *only one* was significantly associated with the later test scores, and it was clearly associated with both of them. That activity was listening to stories. » [WEL09, p. 170]

84. « A more novel evolution-based prediction is that reading comprehension will also be dependent on theory of mind and other folk psychological domains, at least for literary stories, poems, dramas, and other genre that involve human relationships [...] In other words, once people learn to read, they engage in this secondary activity because it allows for the representation of more primary themes, particularly the mental representation and rehearsal of social dynamics. » [GEA07]

strength of our convictions: from there, he searches for a secure foothold for thought). We would also back away from the attitude upon which, even today, the scientists' intellectual approach is based. This is what Feynman explains, who insists that the essence of modern science involves the ever renewed ability of *questioning* the experts' opinions, that is to say not neglecting the possibility that they may be wrong [FEY99][85] ; and even in fact, believing in what he calls the « ignorance of experts »[86].

So actually, since it is essential to *verify* the foundations of the information we have, we need to be able to *rebuild* our knowledge to really understand. This is for example what is done traditionally in mathematics, where knowledge of theorems is not enough, we must also teach how most of the edifice of modern mathematics can be built. This is what Piaget [PIA73][87], Popper [POP77][88], or either, in the field of computer science, Knuth [KNU02][89] say: « You don't really understand something, if you don't try to implement it ».

So we must learn how to try, but how? As we mentioned earlier, compared with explicit methods, teaching methods based on discovery have been criticized, since the majority of controlled experiments that have been conducted show that they are less effective. Nevertheless, a number of questions arise here:

→ *What is it that is measured* ? Essentially, the performance of learning and the cognitive consequences of poorly designed instructional designs, which can even lead to non-learning ; but at the same time, the criticism that has been made also produces an analysis of the causes of these problems, which can be used to solve them. The factors identified are: the role of memory in problem solving, which paradoxically leads to the need to be more explicit in a number of situations; the culturally situated nature of knowledge, which leads to the need to adopt a more holistic approach ;

→ *What do we not (or not well) know how to measure* ? Motivation; creativity, especially in the long run; learning the skills related to solving really hard problems (skills which however, often fall under the very essence of the domain: for example, asking real questions) ;

→ *Which learnings are we talking about* ? Essentially, skills, that are part of a larger whole ; To our knowledge, there is no scientific study of the acquisition of the structure of an *entire* academic field such as mathematics, physics, computer science, or of a field of humanities ; the existing studies only deal with fragments ;

Arrived here, it therefore seems that there are good reasons to think that what the few exceptionally creative personalities we have cited (e.g. Socrates, Descartes, Cartan, Feynman, Grothendieck, Knuth) tell us reflects an essential aspect of the intellectual reality which, until today, has never actually been the topic of a thorough scientific study[90]. In particular, it seems clear is that the way of learning and the activities of these exceptionally creative personalities are characterized by freedom of choice, and inspire in them intense *emotions*.

85. « Science alone of all the subjects contains within itself the lesson of the danger of belief in the infallibility of the greatest teachers in the preceding generation. [...] Learn from science that you must doubt the experts. As a matter of fact, I can also define science another way: *Science is the belief in the ignorance of experts*. » [FEY99]

86. This is obviously a metaphor, what is here referred to is ignorance in the Socratic sense, that is to say, we need to know that to think for ourselves, we must *verify* that we fully understand all the elements from which our thought is built: the starting point of thought is therefore a state where, in a sense, "we know nothing".

87. « To understand is to discover, or reconstruct by rediscovery » [PIA73]

88. « We can grasp a theory only by trying to reinvent it or to reconstruct it, and by trying out, with the help of our imagination, all the consequences of the theory which seem to us to be interesting and important [...] One could say that the process of understanding and the process of the actual production or discovery are very much alike. » [POP77]

89. « Man versteht etwas nicht wirklich, wenn man nicht versucht, es zu implementieren » [KNU02]

90. The only existing study which, to some extent, is an exception, is the Bristol Study [WEL09], but it does not address the acquisition of an academic field. It also gives results that contrast sharply with (but do not contradict) the results of the smaller studies that we have cited.

It is therefore possible that the difference between the exceptionally creative individuals and others rather lies in the exceptional intellectual freedom which, in one way or another, they were fortunate enough to enjoy, and in the *choice* of exercising this freedom. This choice is in fact *very difficult*, and the sociological reality is simply that many people, exceptionally intelligent or not, do not have the courage to do it [STA95][91][MUE11][92].

For all these reasons, it appears to us that despite all the problems it raises, the development of teaching methods really based on discovery is a project of great value, that could be the source of significant progress in many areas, particularly those related to science and technology. We give below a number of criteria that seem to us essential to take into account:

a) It is necessary to effectively scaffold teachings, and therefore to ensure that there are no gaps that make discovery impossible; especially, one should pay attention to *identifying the prerequisites* that the learner must master in order to be able to correctly assimilate what at some point, we want to teach ;

b) It is necessary that the content of the teachings remains well-defined ; what we want is to *unveil the knowledge structure* of a given domain ; so abandoning the idea of a map of the domain to be completely delineated, that is to say, abandoning the idea of a well-structured curriculum, is not an acceptable option: we do not wish to derive toward teachings without form and deprived of any real content ;

c) Discovery is inherently unpredictable: therefore in discovery-based teaching, *the problem of the evaluation criteria is more difficult to resolve* than in "instructionist" teaching methods. Ignoring this issue amounts to skew the evaluation in favour of the learners who incidentally have easy access to the domain's tacit knowledge ;

d) We wish to organize teaching differently, in adopting a less prescriptive approach; to do this, we need to be able to *follow the learners in a more personalized way*, which also means that the *teaching method* must be much more *based on heuristics*, which therefore must have been sufficiently clarified in advance ;

Arrived here, we feel that to implement a discovery-based education, an effective approach could be to start from a database of interesting problems, graded by difficulty and classified by domain and sub-domain, that the professor and the student choose to tackle. Over time, all the typical problems of a given domain should have been understood and resolved by the learner. From there, we must seek how to rebuild the missing elements, especially in terms of identifying methods and rigorous assessment criteria, as well as in terms of efficient heuristics which allow the teacher to guide the learner in his domain discovery process, and in the discovery of the method which best suits her personal intellectual development, and the maturation of her own style.

Finally, placing ourselves from the standpoint of the society as a whole, considering the role that expert knowledge plays, and in light of the knowledge explosion which has been seen since the beginning of the industrial revolution, it is worth asking the question of how the founding attitudes that made these developments possible can be preserved today: is excessive specialization

91. « Creatives are nonconformists. They are willing to defy convention, and even authority to explore new areas and to find the truth. Creatives are persistent. They don't give up when they get frustrated or rebuffed by a problem, they keep at it. Creatives are flexible. They are able to reformulate a problem when facing failure rather than just give up or continue down the same path. [...] Evidence shows that people often choose to follow others even when that means abandoning the truth, and this is especially the case when the conformist strategy is backed by an authority or someone who seems to be in charge » [STA95]

92. « People often reject creative ideas even when espousing creativity as a desired goal. [...] In two studies, we measure and manipulate uncertainty using different methods including: discrete uncertainty feelings, and an uncertainty reduction prime. The results of both studies demonstrated a negative bias toward creativity (relative to practicality) when participants experienced uncertainty. Furthermore, the bias against creativity interfered with participants' ability to recognize a creative idea. These results reveal a concealed barrier that creative actors may face as they attempt to gain acceptance for their novel ideas. [...] If people hold an implicit bias against creativity, then we cannot assume that organizations, institutions or even scientific endeavors will desire and recognize creative ideas even when they explicitly state they want them. [...] the field of creativity may need to shift its current focus from identifying how to generate more creative ideas to identifying how to help innovative institutions recognize and accept creativity. » [MUE11]

inevitable ? For then, how can people *verify*, today as in the past, that the items of information available to them are not mere opinions devoid (or partially devoid) of real basis? In light of what has been said above, the answer to this question seems to be related to particularly two things:

- to the multidisciplinary ability of experts to recognize their own fallibility, and thus to make the effort to produce the good quality explanations which non-experts need to be able to understand (and to *criticize*, from their own experience) the foundations of experts' opinion, or the consequences of their recommendations; moreover, experts also need to know how to make the effort not to confine themselves into the ways of thinking of their own discipline, and to verify that knowledge produced in their discipline is consistent with those developed in other disciplines ;

- to the ability of finding *simplifications*, to the *constantly renewed effort of synthesis* without which the development of knowledge cannot continue. Can one use without understanding ? Yes, to a certain extent. For example, the Ptolemaic system is an extremely sophisticated description of the solar system which, in terms of predictive capacity, is in some cases quantitatively more accurate than the Newtonian description, which is much simpler. But yet the Newtonian model is universally regarded as more correct and more *true*, in fact, precisely because of its greater simplicity, which enables a better understanding of the nature of things. So you could say that in science there is also *a belief in simplicity*[93] which plays an essential role because if the amount of information that one must be able to conceptualize grows without limits, it is clear that sooner or later will come a time when the growth will cease, since human brain capacity is a given that does not change. So far, what made progression possible is a process of *reformulating* existing knowledge, which proceeds in parallel to the process of producing new knowledge, and *makes it possible* to achieve the future evolutions (and revolutions). Furthermore, the evolution of science and technology leads to more or less long-term changes in representation systems. The *language* itself evolves, and can express things using fewer symbols[94] ;

All this shows the fundamental importance of a deep understanding of knowledge, which is not limited to superficial aspects of it. Use without understanding is to some extent possible, but one cannot *invent*, nor *criticize* without understanding, and it seems to us that without a solid education in the discovery, the reformulations and conceptual revolutions mentioned above seem inconceivable. And in gross and in detail, we are convinced that a society that loses such abilities (or which, for reasons related to misunderstood efficiency criteria, lets them become the preserve of a minority) takes the risk of draining the source of the constant renewal (which at one time was called "progress"; what today is called "growth") that enables it to meet the vital needs related to the perpetuation of the conceptual structures underlying its culture.

It is, it seems to us, the beginning of a such an evolution that can be observed today in the world of computing. In light of what has just been said, we better understand the reasons behind it, which seem related to the emergence of an excessive specialization, then of the dogmatism which necessarily arises from a fragmentation of knowledge for which we don't have the answer yet. From this perspective then, the design of discovery-based teaching methods in the field of informatics (provided they also give its rightful place to the state of the art) can help to enable individuals to better manage the complexity of the socio-cultural environment in which they perform their activities, and can also contribute helping the society to overcome the crisis we are currently facing.

93. Another example can be found in the "monster proofs" of important mathematical theorems, for example the proof of the Kepler conjoncture about sphere packing [STE07]; we now know that the conjecture is a theorem, since a proof has been verified by machine: however, this proof, which reduces to a long calculation is not very explanatory, it does not allow mathematicians to really understand why the theorem is true. This is why even today, it is considered unsatisfactory: what we want is a humanly understandable evidence, that is to say, the *concepts*, in the human sense of the term, for understanding why the theorem is true. In the case of the Kepler theorem, the concepts in question have not really been found.

94. For example, in mathematical treatises of the Middle Ages, simple problems that a schoolboy can solve today in a few lines by means of a quadratic equation were stated and resolved using one (or more) *pages* of text written exclusively in natural language; writing the equations of physics by means of complex numbers considerably simplifies their expression, so that you can do many more things using the most concise form thus obtained, etc.

3 Problem solving in informatics

In this section, following Polya [POL45] we first give a model of problem solving in mathematics, that we then compare to the software development activity, firstly considered from a context-free perspective (that is to say, considering the development of just one program), and then from an evolutionary perspective (that is to say, considering what happens when one needs to change the specification of the program). We then identify a number of important features of problem solving in informatics, we put this activity in its context, and we finally show how, in our opinion, it is part of a broader sociocultural movement that gives it its full meaning.

3.1 Programming and problem solving

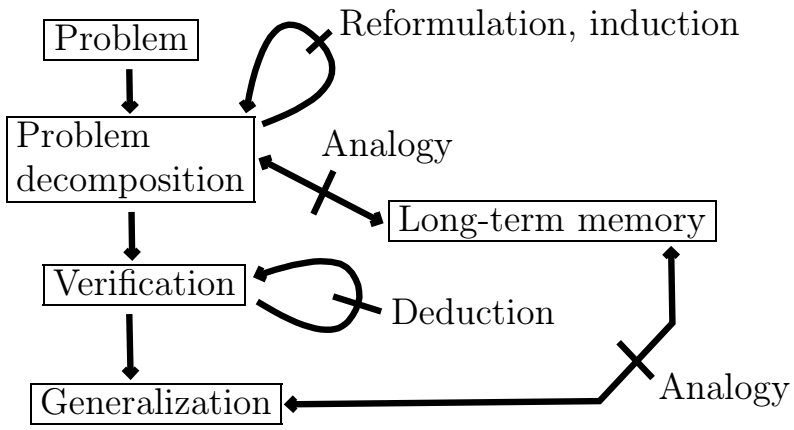


Figure 1 – Solving a problem

According to Polya [POL45], one can pose and solve a mathematical problem using the following four fundamental principles:

1. To understand the problem: this starts with the *vocabulary* used in the formulation of the problem. We must then ask ourselves whether we have understood the problem as a whole, and therefore examine the *feasibility* conditions related to this problem: do we have *enough information* ? What does the problem *depend on* ? Finally, it is important to see how one could examine the problem from different *points of view*, or using different *representations*: Can you *restate* the problem in your own words? Using *diagrams* ? ;

2. To devise a plan for solving the problem: to this end, we can see if we know *similar problems*, or problems related to it. We can also try to see how we could start by trying to solve a *simpler* problem, or either how we could use auxiliary problems, and thus *decompose* the problem into simpler sub-problems ;

3. Set the plan into action: implement the steps of the plan one by one, and *verify* them one after another. At the end, the different elements must have been assembled into a coherent (and *correct*) whole, we must get the solution of the problem, and see *clearly* why this solution is correct ;

4. To examine the solution: having solved the problem is good, but to judge of the quality of the resulting solution, as well as of *the interest of the problem* is even better. Thus we progress, and little by little, we become increasingly skilled in the design of problem-solving plans ;

What stands out at first when you consider problem solving in this way is that the resolution process appears to be very rich, in particular, it is not limited to the purely logical aspect in terms of which the practice of mathematics is often considered exclusively. In fact, it seems clear that a variety of modes of reasoning is used in problem solving. Especially:

- Understanding the problem requires understanding the vocabulary that is used to formulate it, therefore the understanding of *language*. The possible use of various representations also strongly suggests that the raw material from out of which language emanates is not limited to simple text: there are different ways of thinking, that are all important to master ;

- Designing the problem solving strategy calls for a contextualization and for the search of similar solutions, so for *memory* and for *reasoning by analogy*, all things which are strongly related to *creativity* [HOF95]. Furthermore, in addition to calling for analogy (e.g. to formulate the sub-problems), the decomposition of a problem into sub-problems involves *inductive reasoning*, which allows injecting the additional information which is needed to construct the plausible assumptions that are involved in the problem's decomposition (mathematicians know this for a long time, for example see [POI05][95]) ;

- Setting the plan into action, and constructing the solution involve deductive reasoning, which most of the time, reduces not to a simple check: some proofs are of an explanatory nature, some others not[96] (for a more modern formulation of this, see [DOW13]). Furthermore, even when it is simple, deductive reasoning involves a complex implicit know-how which to be acquired, requires a lot of practice, and results from the operationalization of declarative knowledge into a procedural form [AND82]. When the proposed solution can't be built, we must then take a view on the reasons of this and make a diagnosis, an activity that calls for *abductive reasoning* [PEA11] this time ;

- Evaluating the quality of the solution, as well as of the problem's interest, make use of *judgment* skills (aesthetic, in particular) that evoke those which are most commonly attributed to artists. These judging activities seem related to the ability of perceiving some form of minimality, of *elegance*, as well as to mechanisms pertaining to creativity (since incidentally, our assessment of something is not fully intrinsic, it also works by discovering new links with other already available elements) ;

Arrived here (and if you think that the problem-solving model outlined above has more general aspects that go beyond the realm of mathematical reasoning, what we believe), we understand why the thinking of non-experts (e.g. the students described in [KOL05][97]) appears approximative and non-rigorous: this is actually the effect of the fluid appearance that the harmonious development of their intelligence confers to the behavior of individuals. That's why, in our opinion, teaching purely deductive analytical thinking is difficult, because certainly, analytical thinking flies in the face of the intuitions that learners develop in the problem solving activities that they practice in everyday life, which are much more varied in nature. Non-experts therefore tend to not rely on

95. « What is the nature of mathematical reasoning? Is it really deductive, as is commonly supposed? Careful analysis shows us that it is nothing of the kind; that it participates to some extent in the nature of inductive reasoning, and for that reason it is fruitful. [...] A construction only becomes interesting when it can be placed side by side with other analogous constructions for forming species of the same genus. To do this we must necessarily go back from the particular to the general, ascending one or more steps. The analytical process "by construction" does not compel us to descend, but it leaves us at the same level. We can only ascend by mathematical induction, for from it alone can we learn something new. » [POI05]

96. « *Verification* differs from proof precisely because it is analytical, and because it leads to nothing. It leads to nothing because the conclusion is nothing but the premisses translated into another language. A real proof, on the other hand, is fruitful, because the conclusion is in a sense more general than the premisses. » [POI05]

97. « [Students] have different computer-science norms [...] they are tolerant to errors, which are perceived as unavoidable part of the programming reality [...] we believed that students' understand correctness as *relative*. » [KOL05]

this mode of thinking, and to find it difficult and boring [JEN02][98]. It would be wise to ask the question of what could be done to design teachings that *take advantage* of the natural intelligence of individuals, rather than confusing it in this way. The answer to that question should in our view be sought in the direction of designing activities which develop the learners' analytical abilities by presenting them *in context*, that is to say *in combination* with the other modes of reasoning with which in fact they take on their full meaning. From this perspective then, the description of the problem solving process given by Polya highlights the inadequacy of a number of teaching methods in science and technology that are still current today, and also provides avenues that could perhaps be used to overcome this.

We now examine the programming activity in its most simple form, which consists in formulating a specification, in possibly breaking the program to be implemented into sub-programs, and then in implementing and testing. When the program does not work as expected, *debugging* must be undertaken, which consists in identifying the causes of the problem and in fixing the implementation. When the decomposition appears to be unsatisfactory, the program must be *reorganized*, which enables apprehending it more conveniently:

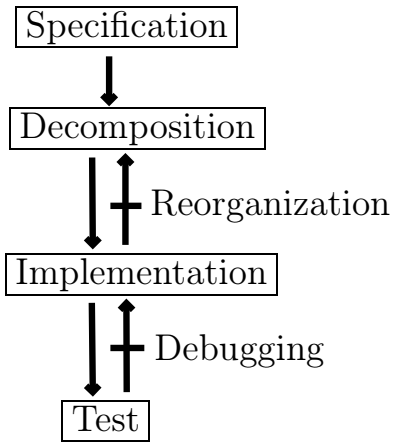


Figure 2 – Problem solving in programming

The programming activity as described above much resembles the problem solving activity that has been described previously. The analogy is not fortuitous in our opinion, since there is a deep connection between algorithms and constructive proofs (the Curry-Howard isomorphism). We have:

- A specification phase, which corresponds to the problem understanding step in the model of problem solving in mathematics given above. One can observe that in the specific case of software development, formulating the specification often takes the form of a *modeling* activity. This does not appear so clearly in the case of the formulation of problems in mathematics, which may have been considered simply for the intrinsic interest that they hold, for the importance of the issues to which they are linked. The programming activity is much more often an engineering activity which consists in building artifacts that are deemed useful, and so informatics seems rather be the science of *designing machines*, for its part ;

- A decomposition phase of the specification and of the program we want to produce: here,

98. « The most common comment is that programming is "boring and difficult". When I ask whether it is boring because it is difficult or difficult because it is boring, they are seldom sure. But they remain adamant that it is both. Teaching a subject that is boring and difficult is a tricky task indeed. [...] It is clear that the students' tried and tested learning styles *do not work* when applied to programming. » [JEN02]

the activity looks pretty close to what the mathematician does, since in particular, it is a question of designing an appropriate *representation* (of the program and of the data it manipulates [BRO75][99]). An essential aspect appears here, which is the fact that a program has a *behavior* ; this reveals another level of representation: in addition to the representation of the problem solving strategy that exists in the mind of the designer and of the form that she chooses to give to the solution (i.e., to the program), there is also the representation of the information that this program will handle during its operation. From this perspective then, a proof is simpler than a program, since it is a static object, it exists outside of time ;

- Two phases, an implementation phase and a test phase of the program, which correspond to the phase of setting the plan into action mentioned above. Here, the additional level related to the program's behavior appears even more clearly: implementing the plan not only consists in verifying that the program's construction plan is feasible and actually makes it possible to write it in a coherent manner, *but also* in verifying that it behaves as expected ;

The analysis that was conducted above reveals a number of similarities and differences between problem solving in mathematics and problem solving in programming, namely:

i. The procedural nature of the obtained solution, which is specific to the programming problem-solving activity. A major difficulty is the *loss of direct manipulation* [BLA02], which arises from the need to reason about the program's behavior, which can only be perceived in a deferred way, in the abstract. Explicit reasoning about procedures indeed seems to be a difficult activity *as such* [AND93][100], and there are convincing arguments in favor of the importance of the role that procedural knowledge plays in human cognition, as well as in favor of the automatic nature of the pertaining processes [SHI77][SLO96]. It doesn't mean that the operation of procedures is always opaque: rather, explicit reasoning about the procedures performed by the mathematician exists, and has been observed [STA09][RIT14][101]. But as we mentioned above, the programming activity takes place *on two levels*: there is firstly a reflexive activity of the programmer who pays attention to the procedures that *she herself* performs to write the program (that activity is similar to what can be observed in algebra, for example) ; and secondly, there is an explicit reasoning activity about the procedural aspects *of the program* under development. The « radical novelty » [DIJ89][102] of programming in this view is that it requires carrying out a reasoning about procedures which must be explicit, comprehensive, and does not tolerate approximation (since the computer doesn't intelligently interprets a program like a human: hence the slightest imperfection can render this program completely ineffective) ;

ii. Program construction, purely considered in terms of handling programs as *texts* written in a formal language, does not seem to fundamentally differ from proof construction as it arises when solving mathematical problems. It seems that writing programs puts into play a form of planning [SOL85] ; the *patterns* that are used to write programs can also be used to read

99. « By the architecture of a system, I mean the complete and detailed specification of [the API]. [...] Representation is the *essence* of programming. [...] The data or tables [...] is where the heart of a program lies. » [BRO75]

100. « procedural knowledge is knowledge people can only manifest in their performance [...] procedural knowledge is not reportable » [AND93, pp. 18-21]

101. « However, at least in mathematical problem solving, people often know and use procedures that are not automatized, but rather require conscious selection, reflection, and sequencing of steps (e.g. solving complex algebraic equations), and this knowledge of procedures can be verbalized [...]. Overall, there is a general consensus that procedural knowledge is the ability to execute action sequences (i.e. procedures) to solve problems. Additional constraints on the definition have been used in some past research, but are typically not made in current research on mathematical cognition. » [RIT14]

102. « Dijkstra argues that [...] programming is what he terms a "radical novelty" [...]. Dijkstra also notes that the "smallest possible perturbation" in a program of one single bit can render a program totally worthless. » [JENK02], about [DIJ89]

programs, and over time acquire a conventional appearance that facilitates the transmission of knowledge between programmers [SOL84]. The ability to build solutions is a skill that develops with practice ; the process of « chunking » that stems from this learning seems to be also linked to the development of abstraction abilities, which operate at different levels in the programming activity [LIS11]. In any case, the ability to shape truly *explanatory* proofs (i.e. which are not limited to simple verifications), seems fundamental, both in mathematics [POI05] and in programming [DIJ71][103] ;

iii. The decomposition, which as we have seen, first consists in designing a representation of the problem which, in particular, determines the shape of the solutions that we are able to consider [HAY77]. In mathematics, the decomposition uses approximate methods of reasoning (inductive and by analogy, in particular), which are of an exploratory nature, and which are used to search for *plausible* assumptions, but plausible modes of reasoning actually irrigate the problem-solving activity in all kinds of areas [POL54][104]. In programming, the decomposition is, with the explicitation of specifications, one of the most difficult points [WIR95][105]. A first essential aspect is that decomposition requires *domain knowledge*, that guide and delineate the choice of possible representations to use in the program, and without which one cannot develop an *overview* of the design currently under development [ADE85][VES85]. Another crucial element is that the decomposition makes possible a good *planning* of the development; without such a "map" of the software to implement, we find ourselves in a situation where we develop various parts of the software, hoping that gradually an overall structure will "emerge" in some way, without really be sure that the process will end. In practice, when the structure of the software under development is not clearly perceived, one runs the risk of *losing control* of the code that one develops [HOU13][106].

However, as we said above, a priori, the decomposition involves plausible reasoning, which is of an exploratory nature. Thus, when the software we must develop is truly *new*, a kind of "bottom-up" development of an approximate nature (which then resembles a lot to what is observed when solving a truly unknown mathematical problem) is inevitable. In such a situation, seeking to accelerate the resolution of the problem by increasing the size of the teams is an approach which often turns out to be catastrophic [BRO75][KNU02][107].

103. « The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence. [...] With the above I touched on five [...] ways by which we can increase the *understandability* of programs. They have to do with the ease with which we can understand what a program is doing, they have to do with the feasibility of correctness *proofs*. » [DIJ71]

104. « We secure our mathematical knowledge by *demonstrative reasoning*, but we support our conjectures by *plausible reasoning*. A mathematical proof is demonstrative reasoning, but the inductive evidence of the physicist, the circumstancial evidence of the lawyer, the documentary evidence of the historian, and the statistical evidence of the economist belong to plausible resasoning. [...] Plausible reasoning is hazardous, controversial and provisional. Demonstrative reasoning penetrates the sciences just as far as mathematics does, but it is in itself (as mathematics is in itself) incapable of yielding essentially new knowledge about the world around us. Anything new that we learn about the world involves plausible reasoning, which is the only kind of reasoning for which we care in everyday affairs. [...] Mathematics is regarded as a demonstrative science. Yet it is only one of its aspects. [...] You have to *guess* a mathematical theorem before you prove it; you have to *guess* the idea of the proof before you carry through the details. You have to combine observations and follow analogies; you have to try and try again. [...] Now, observe that inductive reasoning is a particular case of plausible reasoning. Observe also (what modern writers almost forgot, but some older writers, such as Euler and Laplace, clearly perceived) that the role of inductive evidence in mathematical investigation is similar to its role in physical research. » [POL54]

105. « The most demanding aspect of a system's design is its decomposition into modules. » [WIR95]

106. « When I graduated college I was a lousy programmer. I had a degree and plenty of book knowledge, so I started a side project and spent hours nightly hacking away. Things started off fine, but my coding style wasn't doing me any favors. I copied and pasted, chose poor names, intermixed concerns and regularly created long run-on functions that performed a wide variety of only loosely related tasks. And after a few months, I came to a hard realization: The project was falling down under its own weight. I thought I could get away with being sloppy since I was the only developer. But ultimately I had to admit that I couldn't maintain — let alone comprehend — the mess I'd made! » [HOU13]

107. « But when it comes to the first version of a new system, then I really know no other way than to share it in almost independent parts and to leave each of these parts to just a single person. The individual people need to talk about the interfaces, but not about their own work. If you implement the second generation of a system, you have

Team organization is therefore a difficult point in this context.

Another possible approach to facilitate identification of a good decomposition consists in attempting to use pre-programmed elements (this is the point (a) suggested by Brooks that we mentioned in the first part), but it turns out that in practice, the preexisting software is often rigid and not easy at all to adapt, even when you want to make changes that seem small at first glance. Therefore in such cases, reuse of preexisting components actually proves to be more costly than full rewrite of equivalent components that integrate harmoniously with the software under development [GAR95][108].

However, note that there are still a number of conventional techniques to organize the development, and as far as possible, to facilitate the management of the exploratory aspects of software development projects, particularly the development by layers [DIJ68], the development by stepwise refinement [WIR71], or e.g. the SOLID approach [MAR00]. However (except when we already know the appropriate structure because we have already developed similar software [HAG15][109]), regardless of the method used, the development of a proper decomposition remains an exploratory process, which progresses as the software develops [PEN90][110] ;

iv. Finally, specification, which in the case of programming, takes the form of a modeling activity, is a strenuous activity too, in particular because the object of the modeling is most often linked to the work of a human agent, something that is intrinsically very difficult to model ;

---

a common vocabulary and people familiar with the same things, it's something else. But at the very beginning ... for me, the reason why many projects fail, it's so simple: they try using more than one person. » [KNU02]

108. « All we had to do was put the subsystems together, a task considerably simplified by the fact they were all written in either C++ or C, had all been used in many projects, and we had source code for all of the parts.

A piece of cake? Unfortunately, no. Two years later, after considerable effort (approximately 5 person-years), we managed to get the pieces working together in our first Aesop prototype. But even then, the size of the system was huge (although we contributed a relatively small proportion of our own code), the performance was sluggish, and many parts of the system were difficult to maintain without detailed, low-level understanding of the implementations.

What went wrong? One might argue that we were simply poor systems builders, but we suspect that this experience is not unfamiliar to anyone who has tried to compose similar kinds of software. [...] Our basic conclusion is that many of the hardest problems are best understood as *architectural mismatch* problems. Each component makes assumptions about the structure of the environment in which it is to operate. Most if not all of these assumptions are implicit, and many of them are in conflict with each other. » [GAR95]

109. « If you've previously created a messaging service and you want to build a new messaging service, then you have infinitely more valuable insight than someone who has only worked on satellite power management systems and decides to get into messaging. You know some of the dead ends. You know some of the design decisions to be made. But even if it happens that you've never done any of this before, then nothing is stopping you from diving in and finding your way, and in the end you might even be tremendously successful.

Except when it comes to figuring out how much work it's going to take. In that case, without having done it before, all bets are off. » [HAG15]

110. « The design of software, as with the design of any complicated artifact, presents challenging problems for the developer. One particular source of difficulty results from the fact that the design subtask interacts with other programming subtasks. One reason this occurs is because of the high degree of uncertainty and incomplete information that is typical of a large scale programming project. [...] A second reason for alternating among subtasks is that decisions made in later subtasks, may *alter* decisions made in previous subtasks, necessitating a return to problem understanding or early design phases. » [PEN90]

3.2 Software engineering and programming in the large

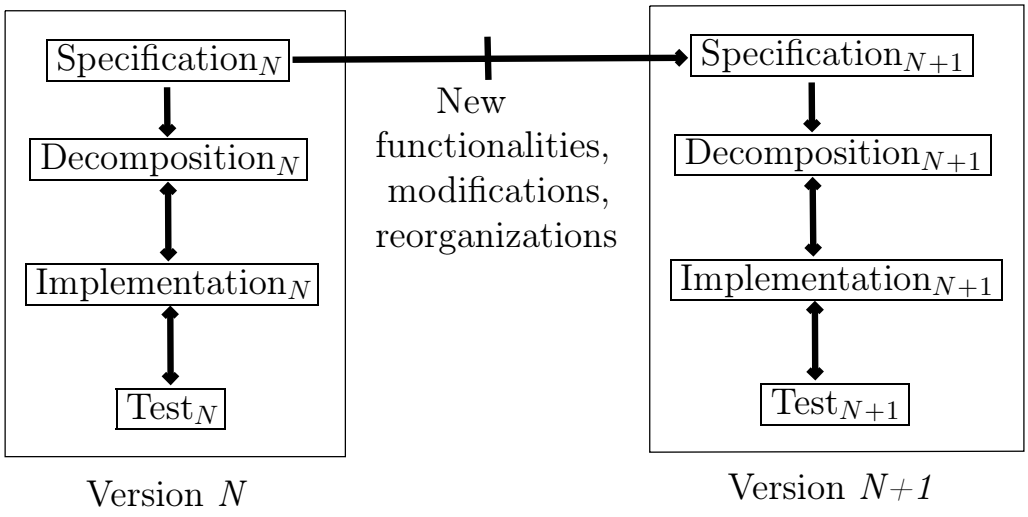


Figure 3 – Software development cycle

We now examine some aspects related to the the software life-cycle and to non-monotonic reasoning in software development ; in particular, software development is a process *that takes place in time*, in at least two ways:

- first of all, as we have seen, in the general case the development of an adequate decomposition is of an exploratory nature, which means that in practice, the software is most often built *in stages* [BAS75]. However, when certain key aspects of the overall design are not anticipated, the architecture that will emerge by default from the development process risks to be inadequate, and it can then turn out to be extremely expensive (or impossible) to make it evolve to obtain the required properties [BOE10][111]. Obviously, it is not always possible to anticipate all problems, and always make sure in advance that the proposed design is feasible: thus, the exploratory aspect that software development presents appears to be irreducible ; that is why one should be aware that it can be necessary to go back on some aspects of the development [BRO75][112], or even in some cases (especially when the design is new), to throw away a version of the software [BRO75][113] ;

- moreover, in the medium term the system *specifications* are also changing, and successive *versions* of the software are produced over time. This leads to reorganizations, or even to reformulations of the implementation and architecture, which can be extremely costly if the changes to be made are not well localized. That is why one must develop code in a modular

111. « The evidence provided across 40 years of data on the degree of increase in software cost-to-fix versus delay-of-fix is that for large projects, the increase from fixing requirements changes and defects during requirements definition to fixing them once the product is fielded continues to be around 100:1. However, this ratio can be significantly reduced by higher investments in early requirements and architecture verification and validation. [...] Small, noncritical projects can also spread their architecting activity across the life cycle via refactoring, but need to watch out for making easiest-first architectural commitments that cannot be easily undone by refactoring » [BOE10]

112. « In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. [...] The discard and redesign may be done in one lump, or it may be done piece-by-piece. But all large-system experience shows that it will be done. » [BRO75]

113. « Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. » [BRO75]

way, and design the modules so as not to give access to items which are believed to be likely to change in the future [PAR72][114]. When their design is successful, such modules do not need to be modified when the software evolves ; when they can also be used as building blocks to develop other applications, one then says that they are *reusable*. To this day, we attach too little importance to modularity and code reuse [GRE04] ; nevertheless, the ability to *guide the software evolution* appears to be a key aim of modern development practices [YAN03][115] ;

Software development thus takes the form of an activity of an irreducibly exploratory and evolutionary nature. In addition to the radically new element pertaining to procedural reasoning identified by Dijkstra in [DIJ89], this *reasoning about the possible development processes* is, in our opinion, the second radical novelty that software development brings.

3.3 What are the possible approaches ?

We often hear that mathematics consists mainly of 'proving theorems'. Is a writer's job mainly that of 'writing sentences' ?

A mathematician's work is mostly a tangle of guesswork, analogy, wishful thinking and frustration, and proof, far from being the core of discovery, is more often than not a way of making sure that our minds are not playing tricks.

Gian-Carlo Rota, in preface to P. Davis and R. Hersh, *The Mathematical Experience* (1981).

"If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected." Such a goal has been a continually recurring theme [...] we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it "computer science."

Implicit in these remarks is the notion that there is something undesirable about an area of human activity that is classified as an "art"; it has to be a Science before it has any real stature.

[...]

Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. [...] The mysterious insights that people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an art.

Donald Knuth – Computer Programming as an Art (1974 ACM Turing Award Lecture)

114. « We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are *likely to change*. Each module is then designed to hide such a decision from the others. » [PAR72]

115. « Twenty years ago, software needed to be corrected occasionally and a new release issued perhaps once a year. We could use the term *maintenance* to imply to that we were working to enable our software to continue to do what it used to do. Ten years ago, software needed a major release with new functionality twice a year, and we used the term *reengineering* to imply that we were adding new user-required functions to the software. Today, software needs to be changed on an ongoing basis with major enhancements required on a short timescale [...]. In this case, the term *evolution* better describes a situation in which maintenance and reengineering are needed so often. » [YAN03]

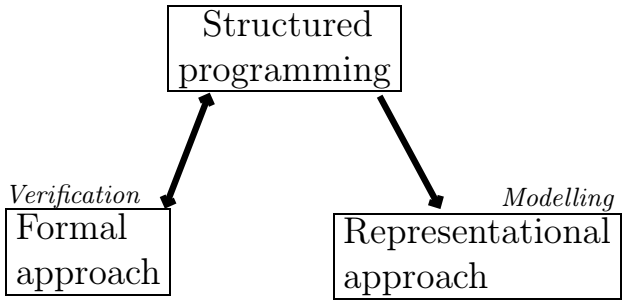


Figure 4 – Possible evolutions of software engineering

Based on the common basis of structured programming, which aims to identify ways to organize programs in a way that facilitates their verification [DIJ69][116], there are in our opinion two major approaches, and two possible evolutions of software engineering: the *formal* approach, which aims to bring software development closer to mathematics, and the *representational* approach, which aims to bring software development closer to areas where modeling of real-world entities (or of virtual entities) plays an important part:

- The formal approach consists in studying programs as mathematical objects, and in looking for ways to write *specifications* that can be associated with the program to ensure that the specification corresponds to the implementation (as in Hoare logic [HOA69], for example) or either be evaluated to check a number of important properties of the program (as in static analysis [WIC95], or as in bisimulation [MIL89]), or even used to generate or to directly execute the program that corresponds to the specification: in this latter case, we are then dealing with *declarative* languages (as in purely functional languages [SAB98], or as in certain languages based on logic programming [HUA11]) ;

- The representational approach consists in focussing instead on ways to develop the specification itself, which is the *essence* of software development [BRO75][117]. The design of a specification often begins[118] with a data model, for which one then develops a sketch of the associated procedures: here we talk about *architecture*, or more generally, about *representation*. In computer science (as in other areas [HAY77]), formulating an adequate representation is the central part[119] of the problem solving process: that is why approaches such as the relational approach [COD70], or the object-oriented approach [DAH67] propose a software development model that appears first as a *modeling* activity (indeed, the first "object-oriented" techniques have been invented in the context of the development of a simulation language, Simula [DAH66]).

  Moreover, the representation that is developed is fundamentally linked to the *mental model* of the application that exists in the user's mind [BRO75][120], especially when the data are interactively perceived and manipulated by means of a metaphor through which the software presents them: the data that exists inside the computer's memory then become

116. « [One should organize] the program structure in connection with a convincing demonstration of the correctness of the program » [DIJ69]

117. « Even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification. » [BRO75]

118. « Sometimes the strategic breakthrough will be a new algorithm [...]. Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious. » [BRO75]

119. « Representation *is* the essence of programming. » [BRO75]

120. « A clean, elegant programming product must present to each of its users a coherent mental model of the application, of strategies for doing the application, and of the user-interface tactics to be used in specifying actions and parameters. » [BRO75]

information, since they represent things that have meaning for the user [TRY09][121]. Thus, software development also takes the form of a *cognitive modeling* activity.

Finally, the most important property is the *conceptual integrity* of the design [BRO75][122], as a result of which the software is easy to conceptualize and remember for the user, and then seems obvious. To exist, conceptual integrity requires some form of simplicity, and so acts as a safeguard that limits the complexity and improves the maintainability of the software [BRO75][123]. Moreover, conceptual integrity can play an important heuristic role, since in general, clients are not designers, and often have difficulty perceiving the consequences of their wishes, and unwittingly, may sometimes choose to jeopardize the elegance or even the consistency of the software. In such cases, conceptual integrity takes precedence, and helps identify acceptable options ;

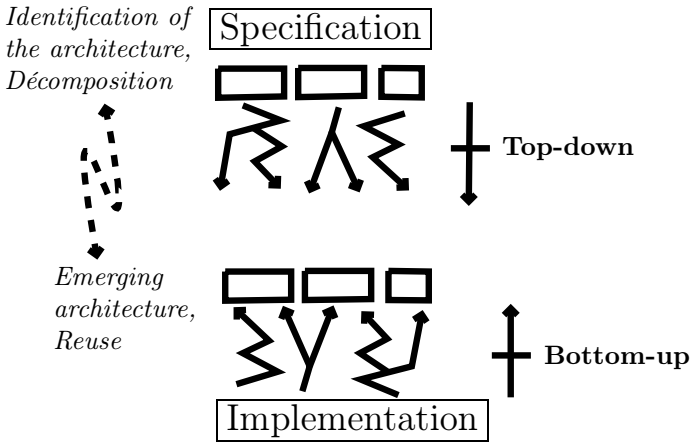


Figure 5 – Top-down vs. Bottom-up approach

As soon as, to achieve conceptual integrity, one separates the development into specification and implementation [BRO75][124], it is natural to develop the software going from the general to the particular (thus following a *top-down* approach) [BRO75][125]. This doesn't exclude possible roundtrips between specification and implementation [BRO75][126], but it often seems better

121. « For a smooth interaction between man and machine, the computer's "mental" model (also the programmer's mental model) and the end user's mental model must align with each other in kind of mind-meld. In the end, any work that users do on their side of the interface manipulates the objects in the code. [...] Both object-oriented design and the Model-View-Controller (MVC) framework grew to support this vision. MVC's goal was to provide the illusion of a direct connection from the end user brain to the computer "brain"—its memory and processor. [...] MVC is all about making connections between computer *data* and stuff in the end user's head. *Data* are the representation of information [...]. [But] they *mean* something only in the mind of the user when there is an interaction between them. The mind of the end user can interpret these data; then they become *information*. » [TRY09]

122. « I will contend that conceptual integrity is *the* most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. [...] The conceptual integrity of the product, as perceived by the user, is the most important factor in ease of use.» [BRO75]

123. « How about the technical director? [...] He provides unity and conceptual integrity to the whole design; thus he serves as a limit on system complexity. » [BRO75]

124. « The separation of architectural effort from implementation is a very powerful way of getting conceptual integrity on very large projects. » [BRO75]

125. « Long before any code exists, the specification must be handed to an outside testing group to be scrutinized for completeness and clarity. [...] The division of system building into architecture, implementation, and realization is an embodiment of these notions; furthermore, each of the architecture, implementation, and realization can be best done by top-down methods. » [BRO75]

126. « [It] does not mean that one never has to go back, scrap the top level, and start the whole thing again as he encounters some unexpectedly knotty detail. Indeed, that happens often. » [BRO75]

to spend time improving the specification rather than trying to solve conceptual problems by modifying the program directly [BOE10][127].

However, as we have already mentioned, the development of software over time is an iterative process, and experience shows that using rapid prototyping methods is an extremely effective way of reducing the cost of developing the specification [BRO86][128]. *Agile* development methods push this logic to the extreme, and consider that the existence of a working version of the software (from where one can in particular discuss with the client) is the most important element to consider in judging the status of a project [AGI01a][129]. Software architecture is no longer thought through an explicit specification drawn up in advance, but rather emerges gradually through a continuous reorganization process (refactoring). Finally, the quality of teams and communication between the different stakeholders of a project are considered key elements [AGI01b][130] (this is well known elsewhere [BRO75][131][WHI13][132]).

Furthermore, in the development of the program's structure seen from a little further, the invention of original reuse options is expected in the medium term (since, through successive reorganisations, there is a need to identify software components that are stable over time[133], thus at least partially reusable), from which unforeseen opportunities to improve the specification will often appear.

In the longer term, during the development of successive versions, one must manage the software *evolution*, and the more time passes, the more likely it is that one will have to question some of the decisions taken. It is therefore clear that a pure top-down approach is not realistic: in the medium term, one is compelled to adopt an iterative approach during which one brings out the design over the course the software evolution[134]. It then comes to a rather bottom-up development type, in which we try to maintain the reusable components that have been developed, and to identify new reuse opportunities. Arrived here, note that the *state of the art* can provide useful leads for devising the most promising architecture at a given time.

In difficult cases, it is the search for conceptual integrity which should rather provide guidance: but when the artifact that has to be produced is truly new, one cannot avoid taking the kind of exploratory approach which is practiced in research projects. Either way, it is essential to realistically identify the specific case in which we find ourselves, to be able to adopt the appropriate approach: otherwise the project may fail not for purely technical, but for managerial reasons [SAU09][135].

127. « The evidence [...] on the degree of increase in software cost-to-fix versus delay-of-fix is that for large projects, the increase from fixing requirements changes and defects during requirements definition to fixing them once the product is fielded continues to be around 100:1. However, this ratio can be significantly reduced by higher investments in early requirements and architecture verification and validation. » [BOE10]

128. « The most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. [...] Therefore one of the most promising of the current technological efforts, and one which attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as part of the iterative specification of requirements. » [BRO86]

129. « Working software is the principal measure of progress » [AGI01a]

130. « We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value: Individuals and interactions over processes and tools [...] Customer collaboration over contract negotiation » [AGI01b]

131. « The quality of the people on a project, and their organization and management, are much more important factors in success than are the tools they use or the technical approaches they take. Subsequent researches have supported that conviction. Boehm's COCOMO model finds that the quality of the team is by far the largest factor in its success, indeed four times more potent than the next largest factor. » [BRO75]

132. « It is notable that information systems project failure is more attributed to organizational and communication related issues than to technological issues [...]. Teams that perform cohesively and purposefully [...] are more likely to successfully identify and overcome uncertainties in a complex adaptive system. » [WHI13]

133. « We propose instead that one begins with a list of difficult design decisions or design decisions which are *likely to change*. Each module is then designed to hide such a decision from the others. » [PAR72]

134. This is indeed what professor Alan Parlis already described during the first founding conference of software engineering in 1968: « Through successive repetitions of this process of interlaced testing and design the model ultimately becomes the software system itself. I think that it is the key of the approach that has been suggested, that there is no such question as testing things after the fact » [NAU69]

135. « Although some of the technology on Mars Climate Orbiter (MCO) had been developed prior to the project's inception and building an orbiter spacecraft was not new, MCO was the first of its kind in integrating all these technologies into one space vehicle. [...] One can thus conclude that MCO was a high-tech project. [...] A high-tech project [requires] long periods of design, development, testing, and redesign with multiple design cycles. [...] However, the pressures, constraints, and challenges to push the boundaries of "faster, better, cheaper" (FBC) on

Overall, there is no single paradigm which would *alone* summarize the essence of software development: it would seem that one should rather look for a *synthesis* of approaches [STR14][136] ; Moreover, descriptions that aim to *abstract* the software in order to make certain aspects easier to apprehend often have the drawback of also making certain key aspects inaccessible [BRO75][137]. This is why software can only be truly understood by means of an approach which does not neglect the understanding of how the underlying machine works, and enables the programmer to understand *multiple* levels of abstraction simultaneously [KNU03][138].

It's the same for development methods: one can't understand everything through a single paradigm ; when the technology taken as a basis for the work is well known (in the case of a "medium-tech" project), we'd better take a top-down approach and seek to specify things in advance ; but in contrast, when it is not (in the case of a "high-tech" project), one would be well advised to adopt a bottom-up approach, and to take the time to explore the possible designs of the artefact that one wants to produce. It is actually the ability to know how to correctly identify the situation in which we find ourselves, and to then adopt the most appropriate approach, which is the key element [SAU09][139].

Finally, note that the preservation of conceptual integrity requires that *few* people are responsible for the development of the specification [BRO75][140][KNU02][141][WIR95][142].

Eventually, it appears to us that software development is not reducible to a set of known recipes that it would suffice to apply, and through which one could always avoid unplanned or problematic situations ; in reality, until today, software development remains an *art*, for at least two reasons:

- because no more in computer science than in mathematics, logic predominates: it is essential to verify things rigorously, but the problem-solving process itself cannot be reduced to the mechanical application of the rules of a logic. There is an irreducible heuristic component, which is what Knuth calls the "mysterious intuitions" by which (among other things) science can be developed, but which does not belong to it [KNU74][143], and which is what for its part,

a limited budget restricted MCO's ability to fully recognize its technological challenges, thus MCO was managed more as a medium-tech project. [...] Based on our analysis we may conclude that the failure of Mars Climate Orbiter was managerial, not technical. In retrospect, management did not (or could not) correctly appreciate the level of complexity, uncertainty, and time pressures involved with MCO. » [SAU09]

136. « In my opinion, a general-purpose language need to support a variety of approaches and combinations of such approaches. [...] I think we are searching for a synthesis of language features, library facilities, and tool support that goes beyond the language wars. We need more communication among the language-design and user communities to help develop that synthesis. It is unlikely to emerge from a single community, a single application area, or from academia alone. » [STR14]

137. « The complexity of software is an essential property, not an accidental one. Hence descriptions of a software entity that abstract away its complexity often abstract away its essence. Mathematics and the physical sciences made great strides for three centuries by constructing simplified models of complex phenomena, deriving, properties from the models, and verifying those properties experimentally. This worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence. » [BRO75]

138. « People who discover the power and beauty of high-level, abstract ideas often make the mistake of believing that concrete ideas at lower levels are relatively worthless and might as well be forgotten. The speaker will argue that, on the contrary, the best computer scientists are thoroughly grounded in basic concepts of how computers actually work, and indeed that the essence of computer science is an ability to understand many levels of abstraction simultaneously. » [KNU03]

139. « When important projects fail, the investigation is often focused on the engineering and technical reasons for the failure. That was the case in NASA's Mars Climate Orbiter (MCO) that was lost in space after completing its nine-month journey to Mars. Yet, in many cases the root cause of the failure is not technical, but managerial. Often the problem is rooted in management's failure to *select the right approach to the specific project*. » [SAU09]

140. « Conceptual integrity [...] dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds. » [BRO75]

141. « But when it comes to the first version of a new system, then I really know no other way than to share it in almost independent parts and to leave each of these parts to just a single person. The individual people need to talk about the interfaces, but not about their own work. If you implement the second generation of a system, you have a common vocabulary and people familiar with the same things, it's something else. But at the very beginning ... for me, the reason why many projects fail, it's so simple: they try using more than one person. » [KNU02]

142. « The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built. » [WIR95]

143. « The mysterious insights that people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an art. » [KNU74]

Polya calls *plausible reasoning* [POL54][144]. Thus, in order to build the rigorously verified artifact (a mathematical proof, a correct software) of interest, it is necessary to know how to guess, to try things, and this approach, which is the essence of the resolution process, is rational, but it is itself of a *non* absolutely rigorous nature: it is know-how, it's art ;

- moreover, problem solving makes only sense from the time we have truly determined *which* problem we want to solve, and *why*. And certainly, as a writer's job is not mainly that of writing sentences, a programmer's job is not limited to writing programs, even correct ones: they still need to be *interesting*. Here, criteria of an aesthetic nature such as the conceptual integrity Brooks is talking about [BRO75][145] seem essential to the formulation of problems for which there are good solutions. And to appreciate the elegance of a solution, the quality of a design, or the interest of a problem, our thinking must involve criteria of a partially subjective nature which belong to the domain of art, or even to the domain of humanities and social sciences. Either way, criteria that are not purely technical in nature but are yet those which make technology *possible* ;

So what are the possible approaches ? It seems that at first, since software development is primarily an art, the acquisition of related skills requires a lot of *practice*: the only way to really understand a software program is to be able to reconstruct it [KNU02][146].

But on the other hand, the art of software development is not exercised in a vacuum: there is a *state of the art* (which we partially outlined in the first part of this paper), and a number of *big ideas* [KER00][147][MEY01][148] that underpin the practice of experts in all areas [BRA99][149]. Once the aspects related to developing elementary modes of reasoning well taken into account, it is these ideas that one should look for in practice, as well as in education.

144. « Mathematics is regarded as a demonstrative science. [...] Yet mathematics in the making resembles any other human knowledge in the making. You have to *guess* a mathematical theorem before you prove it; you have to *guess* the idea of the proof before you carry thru the details. You have to combine observations and follow analogies; you have to try and try again. The result of the mathematician's creative work is demonstrative reasoning, a proof; but the proof is discovered by plausible reasoning, by guessing. » [POL54]

145. « I will contend that conceptual integrity is *the* most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. » [BRO75]

146. « You don't really understand something, if you don't try to implement it » [KNU02]

147. « Universities should be teaching things which are likely to last, for a lifetime if you're lucky, but at least 5 or 10 or 20 years, and that means principles and ideas. At the same time, they should be illustrating them with the best possible examples taken from current practice. » [KER00]

148. « In any engineering discipline, [the tools of the trade] constitute a good part of the professional's bag of tricks. We must, however, look beyond them to the fundamental concepts that have not changed significantly since they emerged when the field took shape some 30 years ago. As in hardware design, *the technology evolves, but the concepts remain.* » [MEY01]

149. « Experts' thinking seems to be organized around big ideas in physics, such as Newton's second law and how it would apply, while novices tend to perceive problem solving in physics as memorizing, recalling, and manipulating equations to get answers. When solving problems, experts in physics often pause to draw a simple qualitative diagram—they do not simply attempt to plug numbers into a formula. [...] Differences between how experts and nonexperts organize knowledge has also been demonstrated in such fields as history [...] Historians [...] knew that no single document or picture could tell the story of history; hence, they thought very hard about their choices. In contrast, the students generally just looked at the pictures and made a selection without regard or qualification. For students, the process was similar to finding the correct answer on a multiple choice test. [...] Experts in other social sciences also organize their problem solving around big ideas [...]. » [BRA99]

4 The HOP! system

All of which is absolutely true. BASIC is actually quite tedious and absurd for getting done the vast array of vivid and ambitious goals that are typical of a modern programmer.

[...]

But all of this misses the point. Those textbook exercises were easy, effective, universal, pedagogically interesting — and nothing even remotely like them can be done with any language other than BASIC. Typing in a simple algorithm yourself, seeing exactly how the computer calculates and iterates in a manner you could duplicate with pencil and paper — say, running an experiment in coin flipping, or making a dot change its position on a screen, propelled by math and logic, and only by math and logic: All of this is priceless. As it was priceless 20 years ago. Only 20 years ago, it was physically possible for millions of kids to do it. Today it is not.

David Brin – Why Johnny can't code

Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release.

[...]

Many languages have something called a macro. But Lisp macros are unique. [...] Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

Paul Graham – Beating the Averages

All this can be achieved because functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts — a general higher-order function and some particular specializing functions. [...] The best analogy with conventional programming is with extensible languages — in effect, the programming language can be extended with new control structures whenever desired.

[...]

Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. [...] Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation. Using these glues one can modularize programs in new and useful ways [...]. This explains why functional programs are so much smaller and easier to write than conventional ones.

John Hughes – Why Functional Programming Matters

It is useful to think of a compiler as simple theorem prover: every diagnosed error is a proof that a certain invariant is violated somewhere. As with any deductive system, the richer the set of axioms, the more interesting the proofs that can be derived from it. [...] There is a continuum between type checking as performed by a compiler, ambitious static analysis, and program verification [...]. The quality of diagnostics produced by these tools is particularly valuable for beginners, and is a revelation for programmers [who don't know them].

Edmond Schonberg and Robert Dewar – A principled approach to SE education

People who discover the power and beauty of high-level, abstract ideas often make the mistake of believing that concrete ideas at lower levels are relatively worthless and might as well be forgotten. The speaker will argue that, on the contrary, the best computer scientists are thoroughly grounded in basic concepts of how computers actually work, and indeed that the essence of computer science is an ability to understand many levels of abstraction simultaneously.

Donald Knuth – Bottom-up education

The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built.

[...]

Reducing complexity and size must be the goal in every step – in system specification, design, and in detailed programming. A programmer's competence should be judged by the ability to find simple solutions, certainly not by productivity measured in "numbers of lines ejected per day". Prolific programmers contribute to certain disaster.

Niklaus Wirth – A Plea for Lean Software

Besides formal courses, any curriculum will include software projects. [...] The standard academic project is not enough for this goal. An essential technique is the long-term project, which students should develop over more than a standard quarter or semester, typically over the course of a year.

Bertrand Meyer – Software Engineering in the Academy

You don't really understand something, if you don't try to implement it.

Donald Knuth – interview about MMIX and the Art of Programming

At the conclusion of the study that was conducted in the previous parts, we now draw a number of conclusions about the nature of the software development activity, and about how in our opinion, it should be taught.

First of all, and following Donald Knuth, we believe that until today, the software development activity remains an *art*[150]. In our view, this implies in particular that:

i. As in the case of understanding a mathematical proof, the understanding of a software system is only obtained by means of a process that aims to reconstruct[151] it. The software development activity must in our view be considered as a *problem-solving* activity, and as such, it is therefore based on the increasingly expert ability of drawing up *plausible reasonings* [POL54], which are irreducibly of an *heuristic* and approximate nature. As a result, learning this art is based on the gradual internalization of implicit know-hows that can only be learnt through *practice*[152] ;

ii. Learning programming must be done by example. This is because first of all, as we mentioned previously, numerous evidence shows that one learns best by working from examples rather than by trying to solve problems without information to guide resolution (this is called "the worked-example effect") [SWE85]. Moreover, it is well known that the cognition of experts is based on the knowledge of a *large number of examples*, which make them able to recognize situations, to classify problems by the major principles that will be used in solution rather than by surface features [CHI81] and, in the longer term, to structure their knowledge around the *big ideas* of the domain [BRA99][153] ;

150. « The mysterious insights that people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an *art*. » [KNU74]

151. « You don't really understand something, if you don't try to implement it. » [KNU02]

152. « The efficient use of plausible reasoning *is a practical skill* and it is learned, as any other practical skill, *by imitation and practice*. I shall try to do my best for the reader who is anxious to learn plausible reasoning, but what I can offer are only examples for imitation and opportunity for practice. » [POL54]

153. « Experts' thinking seems to be organized around big ideas in physics [...]. Differences between how experts and nonexperts organize knowledge has also been demonstrated in such fields as history [...]. Experts in other social sciences also organize their problem solving around big ideas [...]. » [BRA99]

We therefore believe that the teaching of informatics should be conducted by means of *examples*, and that in practice, the learner should practice *reconstructing* these examples: hence the name "Hands-On Programming", that we have chosen for the project presented in this article.

From there how do we do ? The study of the software development activity that we conducted in the previous sections revealed the following key points:

a) Importance of reasoning about the procedural aspects of programs (which according to Dijkstra is the "radical novelty" of informatics [DIJ89]). This presents itself in two forms:

1. Reasoning *about the program's behavior*. Reasoning about the procedural aspects of programs is difficult because in particular, the program's behavior can only be perceived in a second phase, resulting in a *loss of direct manipulation* [BLA02]: programming is « *to have* the computer *do* a task, which will only be performed *later* » [GUI06]. To offset the loss of direct manipulation, the programmer must be able to visualise the *state* of the task at the moment of the program's operation, therefore, he must be able to build a theory allowing him to imagine the future execution of the program, and to possibly *refute* this theory when the program's behavior does not match what was expected. It is therefore necessary that in the first place, the programmer *knows* how to implement such diagnoses (but this approach is not natural: humans have a lax understanding of correctness [KOL05][154], and wrongly assume that the computer correctly interprets their intentions [ROB03][155], when in reality, the absolute rigidity of a computer's behavior blindly executing its program's instructions is precisely what is difficult to understand for the novice [DUB89][156]). Moreover, many empirical studies show that it is essential for the programmer to have a good *mental model of the underlying abstract machine* [PER86][SLE88][DUB89][157] ;

2. Reasoning *about the program* itself. To make it easier to write programs, to our knowledge, there are two approaches: the *modular* approach and the *formal* approach. The modular approach consists in defining ways to organize programs in a way that facilitates their evolution [PAR72][158], as well as their verification [DIJ69][159] (for example, in structured programming, one doesn't use the *goto* so that the blocks always have only one entry and one exit ; furthermore, all techniques that enable factoring out common code make verification much simpler, since one needs to check the common code only once). The formal approach, for its part, consists in looking for ways to write *specifications*, which can sometimes

154. « Specifically, students have different computer-science norms that govern their programming activities. For example, they are tolerant to errors, which are perceived as unavoidable part of the programming reality. Furthermore, thorough testing translates to execute your program for many non-systematically chosen input examples and hope for luck. [...] Professionals' definition for correctness is dichotomous [...]. In contrast, we believed that students' understand correctness as *relative*. » [KOL05]

155. « Similarly, novices know how they intend a given piece of code to be interpreted, so they tend to assume that the computer will interpret it in the same way » [ROB03]

156. « The notion of the system making sense of the program according to its own very rigid rules is a crucial idea for learners to grasp » [DUB89]

157. « [A major issue] is the need to present the beginner with some model or description of the machine she or he is learning to operate via the given programming language. It is then possible to relate some of the troublesome hidden side-effects to events happening in the model, as it is these hidden, and visually unmarked, actions which often cause problems for beginners. However, inventing a consistent story that describes events at the right level of detail is not easy. » [DUB89]

158. « We propose [...] that one begins with a list of difficult design decisions or design decisions which are *likely to change*. Each module is then designed to hide such a decision from the others. » [PAR72]

159. « [One should organize] the program structure in connection with a convincing demonstration of the correctness of the program » [DIJ69]

be used to generate or to directly execute the program that corresponds to the specification, or either be associated to the program in order to ensure that the specification matches the implementation (in particular, *types* can be considered a form of specification, and are a powerful way to verify certain aspects of programs, thereby making their development safer and faster [WIR95][160]) ;

Thus, the dynamic properties of programs being rather difficult to conceptualize [DIJ68b][161] we look for ways to abstract them, and to reason about programs only, or about properties of these programs. But one should keep in mind that incidentally, such abstractions often have the disadvantage of ignoring some key properties of the software [BRO75][162]. Therefore in reality, software can only be truly understood by means of an approach that enables the programmer to understand *multiple* levels of abstraction simultaneously [KNU03][163], which should in our view include the ability to reason about behavioral properties (to be able to assess the *complexity* of a program, for example) ;

b) Software development is of an irreducibly exploratory and evolutionary nature: first, the development of an adequate decomposition is in the general case of an exploratory nature, and in practice, software must be built *in stages* [BAS75] ; on the other hand, usually, successive *versions* of a given software will be developed: consequently, software *evolves* over time. This leads to reorganizations, or even to reformulations of the implementation and architecture, which can be extremely expensive if the developer fails to develop an effective strategy for maintaining the software structure: the ability to *lead the evolution* of the software appears to be a key purpose of modern development practices [YAN03][164]. In addition to the radically new element related to reasoning about the procedural aspects of programs identified by Dijkstra in [DIJ89], this *reasoning about the possible development processes* is the second radical novelty that, in our opinion, software development brings.

But for the programmer to be able to work effectively in this manner, it is necessary that the programming language (as well as the development environment) lend themselves to this: in particular, we need to be able to easily develop and test software components of a reasonable size, and have convenient ways to assemble them. Furthermore, as long as the programmer does not have enough experience to easily indentify the elements that would be needed to structure the process, interactive development remains difficult [PEA83][165]: how

160. « The exclusive use of a strongly typed language was *the most influential factor* in designing this complex system in such a short time. » [WIR95]

161. « My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible. » [DIJ68b]

162. « The complexity of software is an essential property, not an accidental one. Hence descriptions of a software entity that abstract away its complexity often abstract away its essence. » [BRO75]

163. « People who discover the power and beauty of high-level, abstract ideas often make the mistake of believing that concrete ideas at lower levels are relatively worthless and might as well be forgotten. The speaker will argue that, on the contrary [...] the essence of computer science is an ability to understand *many* levels of abstraction simultaneously. » [KNU03]

164. « Twenty years ago, software needed to be corrected occasionally and a new release issued perhaps once a year. We could use the term *maintenance* to imply to that we were working to enable our software to continue to do what it used to do. Ten years ago, software needed a major release with new functionality twice a year, and we used the term *reengineering* to imply that we were adding new user-required functions to the software. Today, software needs to be changed on an ongoing basis with major enhancements required on a short timescale [...]. In this case, the term *evolution* better describes a situation in which maintenance and reengineering are needed so often. » [YAN03]

165. « At this point, someone is bound to object that, in the programming process, it is possible to bypass this step of program development altogether, that one may first make an initial reading of the problem, then sit down at the keyboard and begin composing code to achieve the task. [...] In response to this objection, we allow for the distinction commonly made, and applicable to the cognitive activity of programming, between planning-in-action versus preplanning [...]. [What the programmer] is engaged in as he or she sits down and begins to generate

to *guide* the novice programmer in its activity probably plays a key role in the ripening of her abilities. Finally, tools that facilitate the formation of a *mental model* of the software under development undoubtedly play also an important role in the practice of exploratory and/or evolutionary development, since they facilitate the software comprehension, thus its reformulation ;

c) Representation is the *essence* of the software development process: from this point of view, the most important property is the *conceptual integrity* of the design [BRO75][166], as a result of which the software is easy to conceptualize and remember for the user, and then seems obvious. Furthermore, conceptual integrity plays an important heuristic role in the development process itself, since it requires some form of simplicity, and so acts as a safeguard that limits the complexity and improves the maintainability of the software, and helps identify acceptable options[167]. On the other hand, the representation that is developed is fundamentally linked to the *mental model* of the application that exists in the user's mind [BRO75][168][TRY09][169]: thus, software development also takes the form of a *cognitive modeling* activity.

In practice, to learn how to really develop software, one must *form his judgment*, and develop a personal style that enables capturing the aspects – aesthetic, in particular – that are involved in conceptualizing the conceptual integrity of the designs that one creates. Developing such abilities takes time, but to a large extent, the approach to be taken can be taught: it is the same kind of thing that is taught in art schools, for example[170] ;

In what follows, we will first detail the conclusions that we draw as to the philosophy of the HOP! system, and then spell out this philosophy from the standpoint of the environment first, and from the standpoint of the teaching method thereafter.

programming code without a prior plan is planning-in-action, making decisions as he or she goes about the structure of the program, which evolves as the materials of the program are created. [The] outcomes of such a planning-in-action creative process [have been described] in art, music, and other related domains as a consequence of an iterative series of "conversations" the creator has with his or her partial creations. [...] But to return to programming, although planning-in-action is certainly possible, even sufficient, to produce a program, we expect such a planned-in-action program often to have great costs for the beginning programmer. The reason has to do with the anticipated difficulties of comprehension and debugging when one goes back to try to understand what one has done in writing a program not built with foresight. Of course, for expert programmers the sheer automaticity of many programming projects, since they are able to recall successful plans for similar programs or software systems, will mean that little preplanning will be required for the program code generation. » [PEA83]

166. « I will contend that conceptual integrity is *the* most important consideration in system design. » [BRO75]

167. « A system's ease of use always should be a primary goal, but that ease should be based on an underlying concept that makes the use almost intuitive. [...] The incomprehensible should cause suspicion rather than admiration. [...] Initial designs for sophisticated software applications are invariably complicated [...] Truly good solutions emerge after iterative improvements or after redesigns that exploit new insights, and the most rewarding iterations are those that result in program simplifications. [...] Reducing complexity and size must be the goal in every step – in system specification, design, and in detailed programming. » [WIR95]

168. « A clean, elegant programming product must present to each of its users a coherent mental model of the application, of strategies for doing the application, and of the user-interface tactics to be used in specifying actions and parameters. » [BRO75]

169. « For a smooth interaction between man and machine, the [programmer's mental model] and the end user's mental model must align with each other in kind of mind-meld. » [TRY09]

170. « I think aesthetics can be taught to many. Mostly it is a case of repeatedly showing students good examples and explaining why those examples are good. This takes time, years. I would like for programmers to present portfolios of their work, much in the way artist (including architects) have to. » [STR08]

4.1 Philosophy of the system

Arrived here, we spell out the philosophy of the `HOP!` system, which consists of a software development *system*, and of a *method* to teach this. So there are two things, that it is important to clearly differentiate: the *tools*, whose shape is rather contingent, and the *concepts*, the big ideas in the field, which are more long-lasting, and which truly are the heart of the matter[171]. In the first part, we have seen how the tools complexity exploded, until eventually, this rendered the concepts invisible. In the second and the third part, we analyzed the dynamics of ideas from a technocultural perspective, along with the nature of the domain we are interested in. The question for us is to understand how, in the field of software development, one can design an environment and a teaching method which make the concepts as easy to understand as possible. The analysis of the key points that has just been conducted highlights a number of needs, and reveals a number of questions to which we will now provide answers:

1. The understanding of a given software can only be obtained by means of an approach that aims to reconstruct it. Before a learner can address the problem of the reconstruction of a given software, the building process should have been previously divided into stages of a suitable difficulty. In particular, it is not wise to always let the learner freely explore the problem space [KIR06][172], because that way, even when they manage to actually solve the problem, novices not having enough patterns allowing them to interpret the situations they face experience a *cognitive overload* which prevents them from paying attention to what one wants them to learn: in such cases, it is better to directly provide *worked-examples* [KIR06][173], or even relevant problem solving patterns that will help alleviate the cognitive load [NAD05] ;

2. Learning programming must be done by example. In the longer term, the programmer should be able to structure her knowledge around the *big ideas* in the field. First of all, to make learning by example successful, one should provide *good* examples: that is to say, interesting examples, and also, examples whose relevance is widely apparent to students [LEH00][174]. We must therefore be able to offer a *set* of examples that form a coherent whole, and that can be classified in relation to each other, going from the simplest to the most complex. Furthermore, one must seek how the selected examples illustrate the big ideas in the field, and look for how to illustrate *all* the big ideas in the field by means of examples. Therefore, there are *two* dimensions at play here: a technical dimension, which is related to the complexity of the examples, and a rather technico-cultural dimension, which is related to the importance of the examples with respect to the big ideas in the field (which are not always purely technical ideas) ;

171. « Universities should be teaching things which are likely to last, for a lifetime if you're lucky, but at least 5 or 10 or 20 years, and that means principles and ideas. At the same time, they should be illustrating them with the best possible examples taken from current practice. » [KER00]

« In any engineering discipline, [the tools of the trade] constitute a good part of the professional's bag of tricks. We must, however, look beyond them to the fundamental concepts that have not changed significantly since they emerged when the field took shape some 30 years ago. As in hardware design, *the technology evolves, but the concepts remain.* » [MEY01]

172. « Despite the alleged advantages of unguided environments to help students to derive meaning from learning materials, cognitive load theory suggests that the free exploration of a highly complex environment may generate a heavy working memory load that is detrimental to learning. » [KIR06]

173. « Why does the worked-example effect occur? It can be explained by cognitive load theory,which is grounded in the human cognitive architecture discussed earlier. Solving a problem requires problem-solving search and search must occur using our limited working memory. Problem-solving search is an inefficient way of altering long-term memory because its function is to find a problem solution, not alter long-term memory. » [KIR06]

174. « Because models condense a history of cognitive work into a relatively compact inscription, diagram, or formula, they can render invisible the history of cognitive work that created them. [...] We are not arguing here that children should reinvent civilization, but we have learned the need for careful consideration before providing children with solutions to problems that they do not yet regard as problems. It may be a mistake to give models or other symbolic artifacts to students before the need for them has "ripened" in the classroom and is widely apparent to students; the result may be manipulation of the symbols without understanding. » [LEH00]

3. It is essential for the programmer to have a good *mental model of the underlying abstract machine* ; from the standpoint of education, it would be judicious to try to teach *a model of the underlying system*, perhaps modelled on a suitably simplified machine, as in [KNU99], or even on a simplified operating system, as in [HAN75]. Arrived here, note that the Basic interpreters such as those that equipped the first microcomputers (e.g. the basic of the Commodore 64 [COM84], or the basics that have existed in the MS-DOS environment [GWB86]) actually had the characteristic of providing a simplified but realistic enough programming language, and on the other hand of giving full access to the underlying hardware. The Basic programming language has sometimes been violently criticized for the bad habits it is supposed to give to novice programmers [DIJ75][175], but this kind of criticism seems misguided, since for the reasons we have just said, the Basic environments as they have existed had real pedagogical strengths in practice. Finally, there have been environments which, while having comparable characteristics, also provided the programmer with efficient tools to structure her code (notably Smalltalk [GOL89]).

   But how to "simplify an environment in a realistic way" ? From the perspective of the programming language, one should provide a set of *high-level primitives* that make writing programs easier for novice programmers [PAN00][176] (this is the *closeness of mapping* property of a language or an API). This is well known to the designers of educational programming languages (this criterion is for example explicitly mentioned by the designers of the ABC programming language [PEM14][177] and was already part of the design principles laid down in [SCH78][178]). Generally speaking, the language must therefore be *small*, and *comprehensive* enough to enable performing all the development tasks proposed to be addressed: in particular, when the language is small (and consistent) enough, this facilitates adoption of a teaching style based on *constructing applications* such as the one we seek to develop in the `HOP!` project [AST95][179].

   From the perspective of the environment on the other hand, so that the model of the underlying abstract machine does not remain purely theoretical and may be subject to practical investigations by the learner, it seems important to make the internal evaluation mechanisms explicit: for example, all the elements related to the operation of the heap, or even, how recursive calculations are developed by means of a stack, all this should be *testable* by the programmer by means of a suitable API. Ideally, the environment itself should be almost completely written in the programming language that learners use, and this construction should be *simple*, so that later they can explore the design of the

175. « It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. » [DIJ75]

176. « *Closeness of mapping.* Programming is a process of translating a mental plan into one that is compatible with the computer [...]. The language should minimize the difficulty of this translation by providing high-level primitives that match the operators in the plan, including those that may be specific to a domain the programmer is addressing in the program. If the language does not provide these high-level operators, programmers are forced to compose low-level primitives to achieve their high-level goals. This synthesis has been called one of the greatest cognitive barriers to programming [...]. » [PAN00]

177. « Although [ABC] used mostly classical programming and control structures, such as assignment, procedure and function calls, if and while statements and so on, experience showed that programmers were around an order of magnitude faster at programming than with the classical programming languages it was compared with, such as Pascal, Basic, or C. The main reason behind this was the use in ABC of a small number of high-level data structures. Most programming languages provide a set of low-level data structures, which may then be used to design and build higher-level structures. A key realisation with the third iteration of ABC was that it was the high-level data structures that the programmers needed, and they hardly ever used the low-level structures except to build the higher-level ones. » [PEM14]

178. « [an educational programming language] should be based on two critical and apparently opposing criteria: richness and simplicity - rich in those constructs needed for introducing fundamental concepts in computer programming [but] simple enough to be presented and grasped in a one semester course. » [SCH78]

179. « Courses which use "small" languages, (i.e., languages with minimal syntax like Scheme), tend to be application driven [...]. However, due to the amount of language details in procedural languages such as Pascal, Ada, and C/C++, traditional courses using these languages tend to be language driven. [...] Our experience has shown that applications can successfully be used as a means of motivating language constructs and providing a context for their study. [...] The use of examples which can be incrementally developed is especially beneficial, since numerous language constructs can be presented in the same familiar context. For students this kind of concept reuse is as important as code reuse. » [AST95]

essential elements of the system implementation (thus presents itself the Smalltalk system, in particular). Finally, *reflexivity*, which is the property of a system to be defined by means of itself, is in our opinion an essential concept, and as far as possible, should therefore be illustrated by the design of the system itself ;

4. Knowing how to build a software *on a step-by-step basis* ; from the perspective of pedagogy, the *structuring* of the software examples that the learners are invited to reconstruct can here play a guiding role in learning (we will come back to this), particularly in the development of a series of small algorithms that the learner can develop for herself.

   Moreover, a major problem is that beginners naturally tend to consider the computer as an interlocutor having capabilities similar to those of a human interlocutor: this produces misunderstandings that *prevent* the learner to adopt the approach which is appropriate to program writing [PEA87][180]: it is therefore essential to encourage the learner *not to trust* the code she writes, and therefore, to encourage her to *systematically verify* that the code indeed produces the expected results. From the perspective of the environment, it makes sense to try to provide the most convenient tools possible to actively promote the learning of such an approach. Especially, the system's interactivity, and the ergonomics of the REPL[181] loop play an important role here ;

5. Knowing how to organize programs in a way that facilitates their *evolution*, and knowing how to *lead* the software evolution ; from the perspective of pedagogy, *factoring* and code *reuse* should be taught ; from the point of view of the environment, it is essential to provide tools to effectively support the modular approach. Here, we feel that the following points should be mentioned:

   a) The way the language syntax enables defining *blocks* of instructions has its importance: the syntaxes that delineate blocks by means of start and end markers (as in Pascal or C) are more difficult to use for beginners than those which use indentation (as in Python) [GRA06][182] ; generally speaking, syntax errors are a problem for beginners, that can be solved by means of using *structured editors*: this was done in ABC [MEE91], and more recently in Scratch [RES09] ;

   b) Functions and procedures must first be introduced as *tools to factor out common code*, rather than as a means of expressing recursive algorithms. Indeed, how recursion occurs in traditional programming languages is deceptively simple in our opinion, since it is most often linked to a specific evaluation strategy (the "eager evaluation", i.e. the strategy in which expressions are evaluated as soon as they are associated with a variable [REY99]); but there are many other ways to evaluate programs (including lazy evaluation [HEN76], as well as various forms of abstract evaluation [KIN76][CON93]). So it seems that recursion should rather be seen as a concept that condenses and hides a lot of ideas that are fundamental but very far from obvious: for this reason, if we don't pay attention to the way to introduce it, it may be

180. « By analogy to the natural language listener, the novice programmer assumes the computer can go beyond the information given in the code to a program interpretation. [...] Egocentrism bugs reveal students' beliefs that more of their intention expressing what they want to accomplish is resident in the programming code than is actually present. [...] But do students literally believe that the computer has a mind, can think, and interpret the unstated? Novice programmers will vehemently deny that the computer has these mentalistic traits. Instructors are very good at explaining that computers are dumb and can do nothing but what they are told. However, students' behaviors when working with programs betray their denials, for they act as if the programming language provides more than a mechanistic route to intention-expression. The student's **default strategy** for making sense when encountering difficulties of program interpretation or when writing programs is to resort to the powerful analogy of natural language conversation, to treat the computer as a disambiguating mind that can understand. » [PEA87]

181. REPL ≡ Read Eval Print Loop

182. « [...] for instance, in Java or other languages where program blocks are denoted by curly brackets [...] the compiler does not put any requirements on the structure of the program; one can even write an entire program on one single line, provided that all semicolons and brackets are in the right places. Programs of this kind are nearly impossible to check. However, such horror code cannot be written in Python. We feel that writing structured programs, which are easy to check, follow and maintain is one of the main lessons in any programming course. Using Python, this lesson is taught automatically. » [GRA06]

very poorly understood by learners [LEH00][183]. This is in our opinion the cause of the problems encountered by certain constructivist teaching methods based on the Logo programming language [PEA83][184]: recursion is not at all the kind of concept that one should hope introducing without preliminary explicit instruction. In practice, it therefore seems more appropriate to only introduce recursive programming *after* having given an *explicit* and *complete* model of how the stack operates. One thus avoids using more or less blurred metaphors of the kind "the function calls itself", which force the student to subsequently resort to the development of ad-hoc theories that are often lacking of the necessary explanatory power[185] ;

c) Classes and methods should be introduced as a tool for *describing composite objects* and recursive data structures, as well as for factoring out common code and *encapsulating* functionality. Using inheritance to implement abstract classes made to be specialized later seems to us problematic, because inheritance in object-oriented programming languages does not correspond to a subtyping relationship (this is an unsolved problem that exists in all object-oriented methodologies [SCH08]). In any case, the fundamental concept to be taught is encapsulation, which can be put into practice through the definition and implementation of *interfaces*: the notion of abstract class, used as the specification of the functions exported by a module, here seems to us to be the right tool ;

d) One needs to be able to easily develop and test software components of a reasonable size and have convenient ways to assemble them: so we need a good system of *modules* (from this point of view then, the concept of class as it exists in object-oriented programming is incomplete, since a module is also an element that should be able to exist independently, and possibly play the role of a compilation unit). Here it seems to us that the best approach consists in making the simplest choices, guided by a well-designed and well-established system such as Modula [WIR80], for example ;

e) Code *reuse* should be taught: in our opinion, this can only be done by means of long-term activities, which include the development of *several versions* of a given software ;

f) The use of *version management* tools such as SVN [PIL04], or GIT [CHA09] should be taught ; as far as possible, these tools should be *integrated* into the environment ;

6. Looking for means to write *specifications* ; from the perspective of pedagogy, it might be judicious to teach a relatively informal but rigorous method to write specifications *on paper*, in natural language and using various kinds of *diagrams*. To our knowledge, this kind of thing is relatively unexplored, but there are still effective ways to draft the various kinds of documents that accompany software, that all programmers should know.

183. « Because models condense a history of cognitive work into a relatively compact inscription, diagram, or formula, they can render invisible the history of cognitive work that created them. [...] It may be a mistake to give models or other symbolic artifacts to students before the need for them has "ripened" in the classroom and is widely apparent to students; the result may be manipulation of the symbols without understanding. » [LEH00]

184. « We find that little thought or research has been directed to the important problem of articulating intermediate levels of computer programming mastery. [...] For example, 5-year-olds who can get a graphics cursor to work in the Logo programming language are called "programmers", conveying the popular assumption that they have come to understand the logical operations ingredient to a program's flow of control, and are capable of all the cognitive subtasks of programming. [...] To take one example [...] we have found that child novices frequently adopt a systematic but misguided conception of the manner in which control is passed between Logo procedures. Many children believe that placing the name of the executing procedure within that procedure causes execution to "1oop" back through the procedure when, in fact, what happens is that control is passed to a copy of the executing procedure. » [PEA83]

185. As we said above, it is essential that the programmer has a good mental model of the underlying abstract machine: especially in the case of complex (and problematic from a pedagogical point of view) primitives such as the function call, or the memory allocation, this means we must have the opportunity to always know *exactly* what happens when an instruction is executed. The model should therefore not be a simple metaphor which would leave a certain number of elements implicit: it must instead be fully explicit, that is to say, *comprehensive*, and furthermore, be as simple as possible.

From the perspective of the environment, one should seek ways to add language features enabling program *verification*. This can be done in two ways: by means of *types*, as in Pascal [WIR75] and Modula [WIR80], or even with type inference, as in ML [MIL78] (in the context of an educational programming language, this technique has been used in B, a predecessor of the ABC programming language [MEE83]). On the other hand, one can try to introduce *assertions* into the language: it is the basis of *design by contract* in Eiffel [MEY97] ; in the context of an environment dedicated to learning programming, the introduction of formal methods based on invariants, and of a tool that compiles formal diagrams into Python was implemented in [BAC07].

In total, verification tools can facilitate the learning of rigour, but *do not replace* the clear vision that one we must have of the program and its specification, and of the underlying machine as well ; in principle, they are therefore not essential to learnings. Simply, in practice, there are too much repetitive verifications to make when developing programs for human error to always be able to be avoided: that's why verification tools become important, and as such, should be known and have been practiced by programmers ;

7. The programmer should be able to visualise the *state* of the task when the program will be running, and to *refute* her a-priori assumptions when the program's behavior does not match what was expected. From the perspective of pedagogy first of all, debugging is a special case of *diagnosis* [PEN90][186], and this approach is not natural [KOL05] ; furthermore, to effectively debug, one needs to have a good mental model of the problem one is trying to diagnose [PEN90][187]. It therefore seems sensible to insist on this, and to explicitly describe the diagnostic approach to students, and to make them practice it.

   From the perspective of the environment on the other hand, of course we need a debugger, but the importance of having a good mental model of the problem we are trying to diagnose leads us to think that furthermore, we also need an environment which is operating in a sufficiently *transparent* way ; in this respect, the environments based on a programming language that is compiled and evaluated in a very sophisticated way pose difficult problems, because in such a context, an effective understanding of the bugs that one seeks to diagnose then requires an understanding of the underlying translation process: the more the programming language provides opportunities for the compiler to reformulate the program intelligently, the more it may be difficult for the programmer to interpret its behavior (or the little that it is able to perceive). When the programming language semantics (or the semantics of an abstraction of this programming language, e.g. typing information) is simple enough, one can take advantage of this to implement highly effective debugging techniques (e.g., algorithmic debugging [SHA82]). But evaluation methods such as lazy evaluation for example, require writing debuggers that reformulate the evaluator's trace well enough, which poses complex technical problems (in particular, one may need to recalculate parts of the trace by reevaluating a subset of the program only ; yet it's not trivial to recalculate intermediate results regardless of the original context, since in lazy evaluation, the evaluation strategy of a part of the program depends on the state of the overall calculation [NAI95][NIL98]). Finally, the more the decision process implemented by a program (such as an interpreter, or even an expert system) is sophisticated, the more it becomes difficult to use direct observation to understand the reasons for the results it produces: in the general case, one needs *explainers*, which are programs that reformulate the trace and synthesize the information the user needs by means of the contents of the trace, of informations on the program structure, and of domain knowledge [WEI80][WIC92].

   To our knowledge, the problems posed by the development of debuggers for programming languages having sophisticated semantics still remain little studied today, and overall are poorly resolved: it therefore seems reasonable to take this into account, and to choose the simplest and the most conventional execution mechanisms for the programming language on which the `HOP!` system is based ;

186. « Debugging is a diagnostic task, similar to other diagnostic tasks such as medical diagnosis and electronic troubleshooting. » [PEN90]

187. « Debugging largely involves understanding what a program *is* doing and what it is *supposed* to be doing. [...] As a consequence, experts appear to develop a fairly complete mental representation (model) of the program and to understand the possibilities of error as causal models of error in this context. » [PEN90]

8. We should adopt an approach that enables the programmer to grasp *multiple* levels of abstraction simultaneously, and to know how to visualise the operation of a program in the wider context of the *system* that it constitutes with the environment in which it is plunged. Understanding the different *layers* that constitute this system is essential here: there is first the machine (or the network of machines) running the program, then the operating system, then the programming language and the different translation levels that it implements, then the program itself and its architecture. Each time one performs an action on the program (either to understand it or to make it evolve), one has to know how to place oneself at the *right level* of abstraction, and to conceptualize the appropriate representation.

   From the perspective of pedagogy, all this should therefore be firstly explained, and the techniques that allow expressing the architecture of systems by means of abstract and higher level concepts should only be introduced *afterwards*, and be presented as the answer to the problem of conceptualizing the structure and the functioning of complex systems. In substance, abstraction is used to capture the essence of the design ; this can be done in two ways: by means of a *decomposition* [KNU02][188], possibly hierarchical [BRO75][189], or by means of *languages* allowing to ignore a certain number of details, and therefore, to better express the most important (because whatever the level of abstraction at which it operates, on average, a programmer still writes the same number of lines of code in a given time: this is "Corbató's law" [COR69][190]).

   From the perspective of the environment, we need tools such as functions, classes and modules about which we have already talked, which are necessary for implementing the traditional decomposition methods (i.e. the development by layers [DIJ68] and the development by stepwise refinement [WIR71]). The techniques used for developing systems that rely on language-related elements are for their part more varied: firstly, there are those that rely on a fine tuning of some control flow features to which, usually, the programmer does not have access to (as in CLOS [KEE88][KIC91], or as in Flavors [CAN82]) ; then there are those who rely on the extension and reuse of existing structures (such as the definition of classes that emulate builtin Python types like containers or functions [KET12], or the definition of new devices in UNIX [THO78]) ; there are also all the techniques which, in one way or another, affect the behavior of the underlying language (like the AOP [KIC97]), or enable extending the language (like the templates in C++ [VEL95], or the higher-order functions in functional languages [HUG90][191]). Finally, there are all the techniques by means of which one can develop additional language layers (such as Lisp macros [HAR63], or DSLs [MER05]).

   Arrived here, let's note that all these "advanced" techniques have something in common: they require knowledge of a more or less extended part of the inner workings of the underlying system, or even reveal the nature of some or all of the techniques which are needed for the implementation of this system. Thus, the programmer who knows these techniques is encouraged to visualise the functioning of the program under development in the larger context of the *system* in which it is immersed: for us, it is this very approach that matters, more than the sum of the more or less esoteric techniques that a programmer can know.

188. « But when it comes to the first version of a new system, then I really know no other way than to share it in almost independent parts and to leave each of these parts to just a single person. [...] for me, the reason why many projects fail, it's so simple: they try using more than one person. » [KNU02]

189. « For quite large products, one mind cannot do all of the architecture, even after all implementation concerns have been split off. So it is necessary for the system master architect to partition the system into subsystems. The subsystem boundaries must be at those places where interfaces between the subsystems are minimal and easiest to define rigorously. Then each piece will have its own architect, who must report to the system master architect with respect to the architecture. Clearly this process can proceed recursively as required. » [BRO75]

190. « It's our experience that regardless of whether one is dealing with assembly language or compiler language, the number of debugged lines of source code per day is about the same! » [COR69]

191. « All this can be achieved because functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts — a general higher-order function and some particular specializing functions. [...] The best analogy with conventional programming is with extensible languages — in effect, the programming language can be extended with new control structures whenever desired. » [HUG90]

It therefore seems to us that here, the right approach consists in initially achieve an implementation of the `HOP!` system which is as transparent as possible, and then, from the techniques and tools that will have been used, choose those that seem most appropriate to illustrate the design and development of a complex system by means of a multi-level approach based on the development of languages and tools ;

9. How to *guide* the novice programmer in its activity plays a critical role in the ripening of her abilities ; from the perspective of pedagogy, one can guide the learner in two different ways:

    a) On the one hand, one can guide the learner *in her* interactive and/or incremental *implementation effort* of a given software:

        i. the first question that arises here is that of identifying the *things to say explicitly*, which are the things that the learner may have trouble to easily rediscover all by herself. For example, explain that one should not trust the code one writes, and that for this reason, software can only be developed by stages, all of which must be rigorously verified ; or even, show the specific functioning of recursive function calls, or of the dynamic memory allocation ; highlight the importance of conceptual integrity in software design, etc. In short, the things to say explicitly are the *big ideas of the field* and the significant designs of the state of the art, as well as a number of *difficult or non-intuitive issues* that one needs to know ;

        ii. Secondly, we must also identify the elements related to know-how that must be internalized, and which are mainly learnt by doing, for example how to decompose a software into modules and write good APIs, how to debug judiciously, etc. For these kind of things (which are those related to the good command of various modes of *plausible reasoning* in the field), one should provide suggestions, but these suggestions are mainly elements of *style*, and their good command requires effort and *personal* choices from the student, these are not things that can be taught: what can be taught are elements from which the learner will build her own personality, her personal creativity in the art which she practices ;

        iii. Third, defining the *stages* of the software construction one is working on, which can be more or less open, more or less large, etc., is essential to the progressive acquisition of concepts: ideally, these steps should be *adapted* to the abilities of the learner at a given time ;

    b) on the the other hand, one can guide the learner *in the choice of the software she chooses to make* over time; this raises the question of how to *classify* these software, by progressive technical difficulty, as well as in regards to the big ideas of the field ;

    From the perspective of the environment, it is conceivable to support some of the elements related to didactic and pedagogy, in particular by means of:

    - a *dictionary of concepts*, which defines what we mean when we guide the learner ;
    - a module to continuously *evaluate* the knowledge and skills of the learner, which enables deciding how we can interactively vary the granularity of steps in the construction of the software she's working on ;
    - *interactive documents*, which are interactively configured by the evaluation module ;
    - a system of *indexing and structuring* the database of examples, especially by progressive technical difficulty, and by concept ;

10. The most important property is the *conceptual integrity* of the design ; it is therefore essential to show significant examples from the state of the art, and to explain why the conceptual integrity of these software is remarkable ; moreover, the medium-term objective must be that students *design software themselves* ;

Software development also takes the form of a *cognitive modeling* activity ; therefore at some point, students should be working on projects that give them the opportunity to work with clients, and to practice *requirements elicitation*, along with *writing software specifications* ;

In practice, to learn how to really develop software, one must *form his judgment*, and develop a personal style that enables capturing the aesthetic aspects that are involved in the design approach for software development ; this means that in practice, learners must develop *many projects*[192] ;

Based on the elements of answer that stem from the analysis that has been conducted, we now summarize the overall philosophy of the HOP! system ; from the perspective of pedagogy:

- P1: One learns by reconstructing. But the building process must previously have been divided into stages of an appropriate difficulty ; one should not always let the learner freely explore the problem space: in many cases, it is better to directly provide worked-examples, or even relevant problem solving patterns that will help alleviate the cognitive load ;

- P2: One should provide good examples: that is to say, interesting examples, and also, examples whose relevance is widely apparent to students ; we must propose a set of examples that form a coherent whole, and that can be classified in relation to each other, going from the simplest to the most complex ; one must seek how to illustrate all the big ideas in the field by means of examples ;

- P3: One should try to teach a model of the underlying system ; the programming language should provide a set of high-level primitives that make writing programs easier for novice programmers (closeness of mapping) ;

- P4: It is essential to encourage the learner not to trust the code she writes, and therefore, to encourage her to systematically verify that the code indeed produces the expected results ;

- P5: Factoring out common code, as well as code reuse should be taught ; especially:

  → Functions and procedures must first be introduced as tools to factor out common code, rather than as a means of expressing recursive algorithms ; it seems more appropriate to introduce recursive programming only after having given an explicit and complete model of how the stack operates ;

  → Classes and methods should be introduced as a tool for describing composite objects and recursive data structures, as well as for factoring out common code and encapsulating functionality. The fundamental concept to be taught is encapsulation, which can be put into practice through the definition and implementation of interfaces ;

  → Code reuse should be taught: in our opinion, this can only be done by means of long-term activities, which include the development of several versions of a given software ;

  → The use of version management tools should be taught ;

- P6: One should teach a relatively informal but rigorous method to write specifications on paper, in natural language and using various kinds of diagrams ;

- P7: One should insist on the fact that it is important to be able to refute his own a-priori assumptions when necessary, and that this approach is not natural ; there is a logic of diagnosis (abductive logic) ; one should explicitly describe the diagnostic approach to students, and make them practice it ;

192. « I think aesthetics can be taught to many. Mostly it is a case of repeatedly showing students good examples and explaining why those examples are good. This takes time, years. I would like for programmers to present portfolios of their work, much in the way artist (including architects) have to. » [STR08]

- P8: One should adopt an approach that enables the programmer to grasp multiple levels of abstraction simultaneously, and to know how to visualise the operation of a program in the wider context of the system that it constitutes with the environment in which it is plunged ; here, abstraction is used to capture the essence of the design ; this can be done in two ways: by means of a (possibly hierarchical) decomposition, or by means of languages that will be used to implement the software while then writing much less lines of code ;
- P9: To guide the learner, one should:
  - → Identify the things to say explicitly (the big ideas, the difficult issues), along with the elements related to know-how, which are more implicit ;
  - → Define the stages of the construction of a given software, which can be more or less open, more or less large, etc. ; these steps should be adapted to the abilities of the learner at a given time ;
  - → Guide her in the choice of the software she chooses to make ; these software should be classified by progressive technical difficulty, as well as in regards to the big ideas of the field ;
- P10: One should show significant examples from the state of the art ; in the medium-term, students should be working on projects that give them the opportunity to practice requirements elicitation, along with writing software specifications ; the learners should develop many projects (both large and small) ;

From the perspective of the environment, we have:

- E3: The programming language should be small, and comprehensive enough ; we must make the internal execution mechanisms (such as the function call mechanism) explicit ; the environment itself should be almost completely written in the programming language that learners use, and this construction should be simple ;
- E4: One should provide the most convenient tools possible to actively promote the learning of the incremental approach ; the system's interactivity, and the ergonomics of the interactive toplevel loop play an important role here ;
- E5: The environment should provide tools to effectively support the modular approach ; especially:
  - → the programming language should use indentation to format blocks, as in Python ;
  - → fonctions, classes and methods are needed, but not necessarily inheritance ;
  - → a good system of modules is needed ;
  - → as far as possible, version management tools such should be integrated ;
- E6: One should seek ways to add language features enabling program verification, by means of types, by means of assertions, or even by means of program-proving techniques ;
- E7: The environment should operate in a sufficiently transparent way, and the execution mechanisms for the programming language on which the system is based should be the simplest and the most conventional possible ;
- E8: Achieve an implementation of the `HOP!` system which is as transparent as possible, and from the techniques and tools that will have been used, integrate those that seem most appropriate to illustrate the design and development of software by means of an approach based on the development of languages ;
- E9: Support some of the elements related to didactic and pedagogy: guide the learner by means of an evaluation module and of interactive documents that are indexed in an appropriate way ;

4.2 The HOP! environment

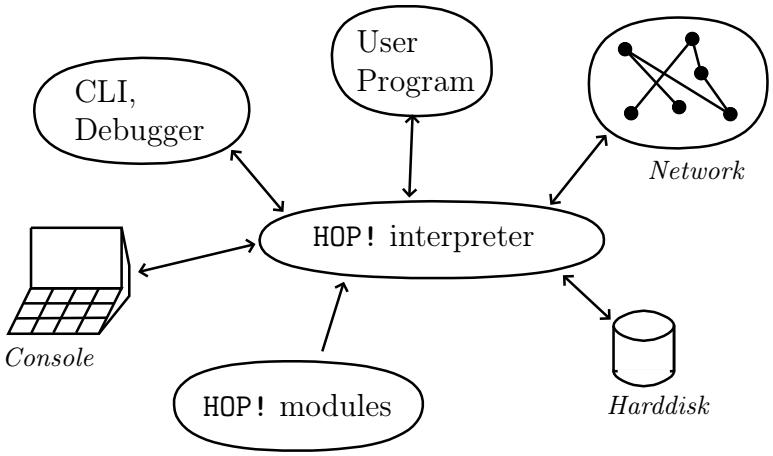


Figure 6 – The HOP! development environment

The HOP! environment provides everything one needs for developing modern applications, in the form of a programming language designed to be *as simple as possible*, while giving the opportunity to its users to develop *realistic* software by means of *small* APIs[193] ; especially:

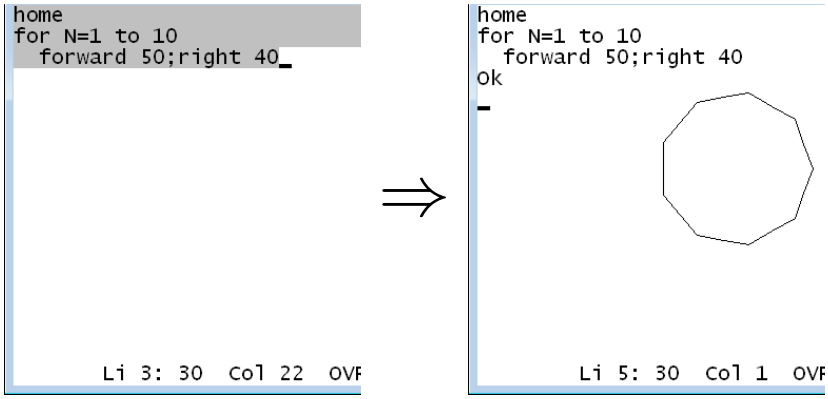
- objects, that are either *atoms* (numbers, strings), or *composite objects* (tables and records) ;
- the usual *instructions* (`for` and `while` loops, `if` statement) ;
- *functions*, the stack being a first-class citizen in the system ;
- *threads* ;
- a very simple system of *modules* ;
- an API for manipulating the *graphical* console (to display graphics and text, and read the mouse, the keyboard and the screen contents, especially) ;
- an API to access the *filesystem* provided by the underlying operating system ;
- an API to access the *network* by means of sockets ;
- an API for *debugging*, which enables checking the contents of the memory at various levels of detail, tracing programs and posing breakpoints ;

First and foremost, the adopted philosophy consists in showing that one can write interesting software in *few* lines of code, and secondly, in providing the means to move from the interactive development of small programs directly at the console to a more mature practice of modular programming in which the current program under development is divided into modules that are edited asynchronously using an external editor.

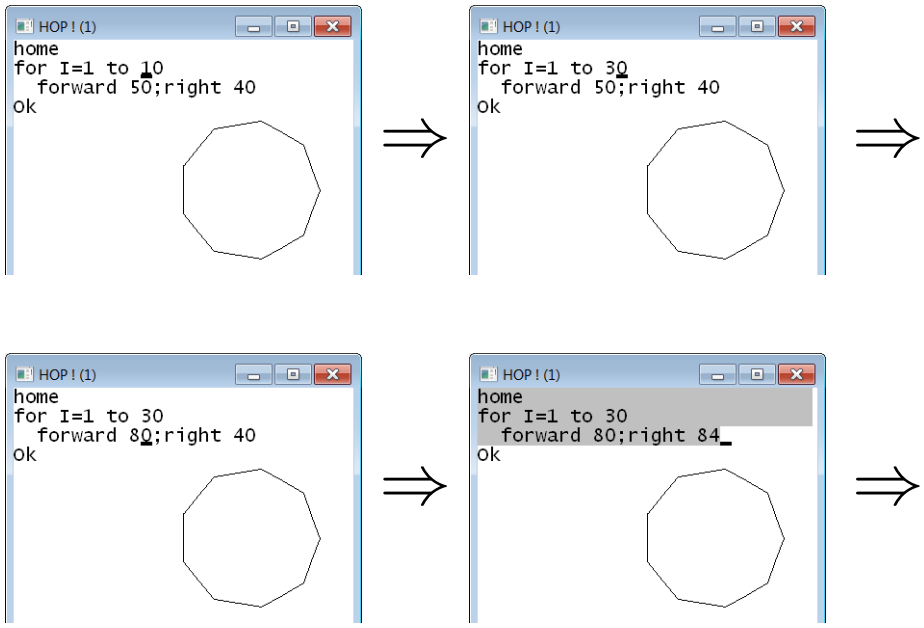
The system is made to be very easy to use to enter interactive commands and/or code snippets (including instructions spanning multiple lines), to obtain the program's output in the same window, and to modify the code and observe the difference in the program's output, and thus be in a position to develop small programs step by step in an incremental manner.

193. For a more detailed description, see `http://hop-project.org/HOPTutorial.pdf`

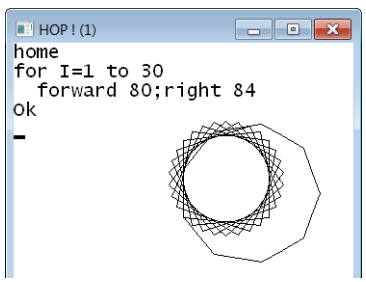
To begin with, here's an example taken from the activities based on the Logo turtle that we propose[194] ; the initial program below draws a regular polygon with nine sides:



Now let's modify the program directly at the console using the following sequence of editing operations:



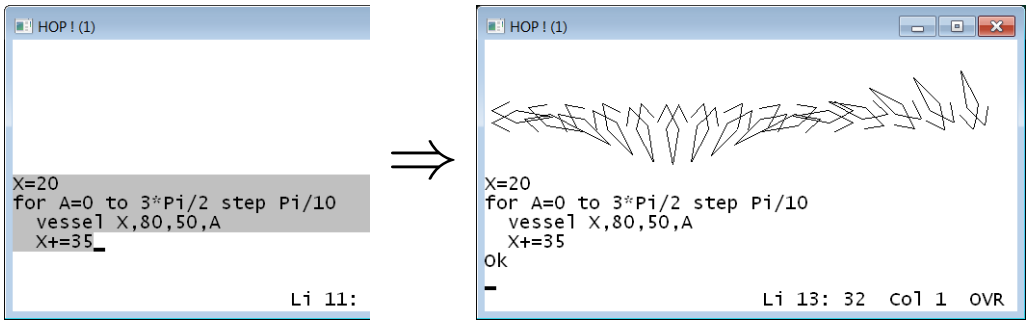
We obtain:



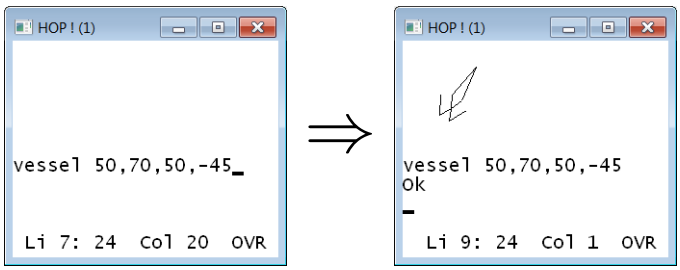
194. For more informations, see `http://hop-project.org/lectures.html`

We now illustrate the "hands-on" approach which is used in the HOP! project by means of a series of examples taken from the activity of reconstructing the video game *Asteroids* that we propose[195].

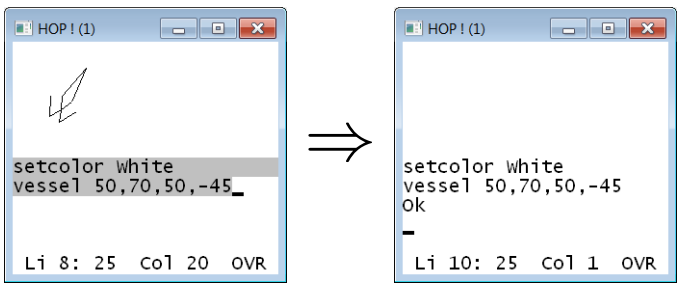
First of all, the *primitive functions* (like the commands that we have just used to manipulate the Logo turtle, i.e. the commands `home()`, `forward()` and `right()` that appear in the above screenshots) which are needed to rebuild a given software in the context of such an "hands-on" activity are provided in a pre-programmed module ; a convenient way to begin the activity is then often to start by exploring the functionalities of these primitive functions. For example, the functioning of the `vessel()` function, which draws the spaceship that is manipulated by the player, can be illustrated by means of small programs like this one:



Once the functioning of the primitives has been assimilated by the learner, one can then introduce other examples that are more specifically intended to illustrate the operation of the algorithms used in the program that we have in hand, seen *from the inside* this time ; thus, to illustrate the simple animation technique we use in our implementation of the *Asteroids* game, we first show a command that displays the vessel in a given position:

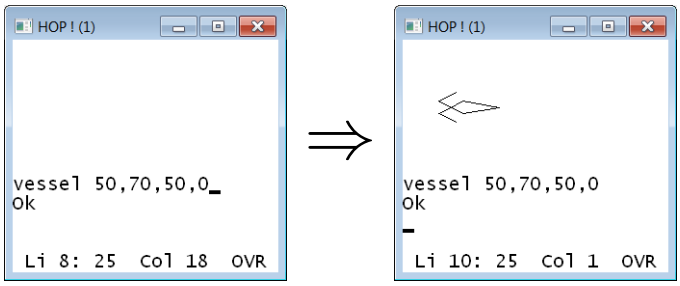


We then show how to make it disappear for a short time by repeating the same command, while redrawing in the background color (i.e., in white, using the "`setcolor White`" statement):

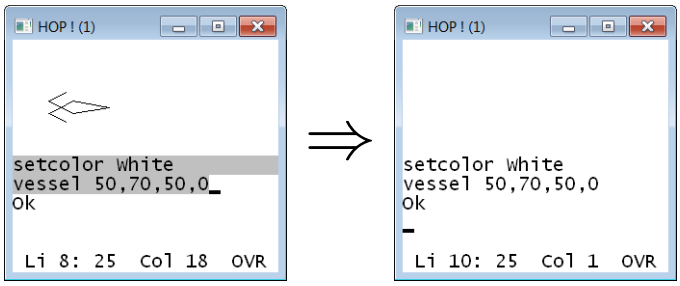


195. See `http://hop-project.org/lectures.html`

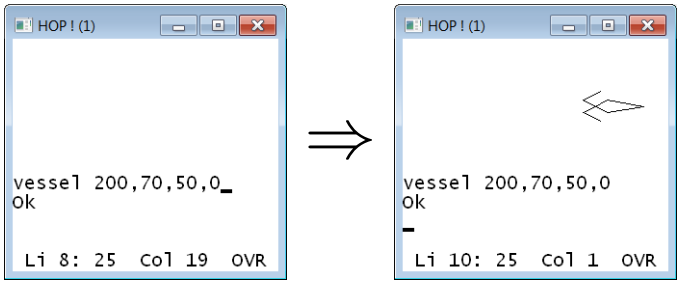
To rotate the vessel, we then draw it in an horizontal position (by changing the value of the fourth parameter of the `vessel()` function):



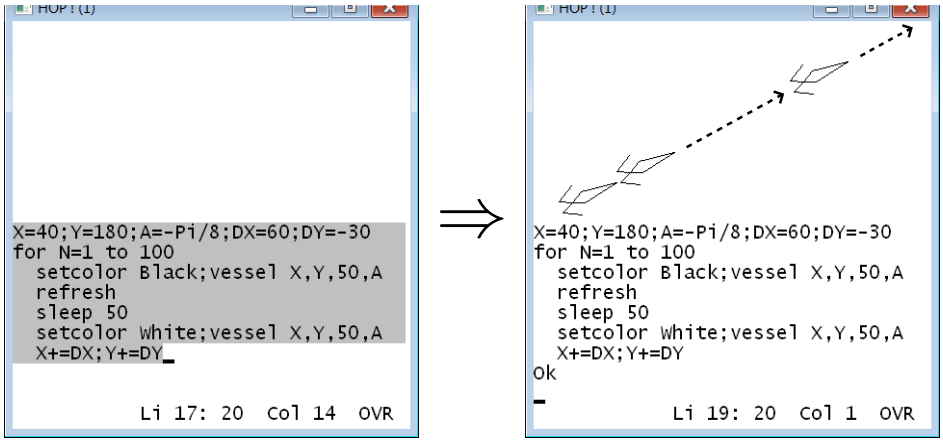
To move the ship forward, we proceed in the same way ; it is first redrawn in the background color:



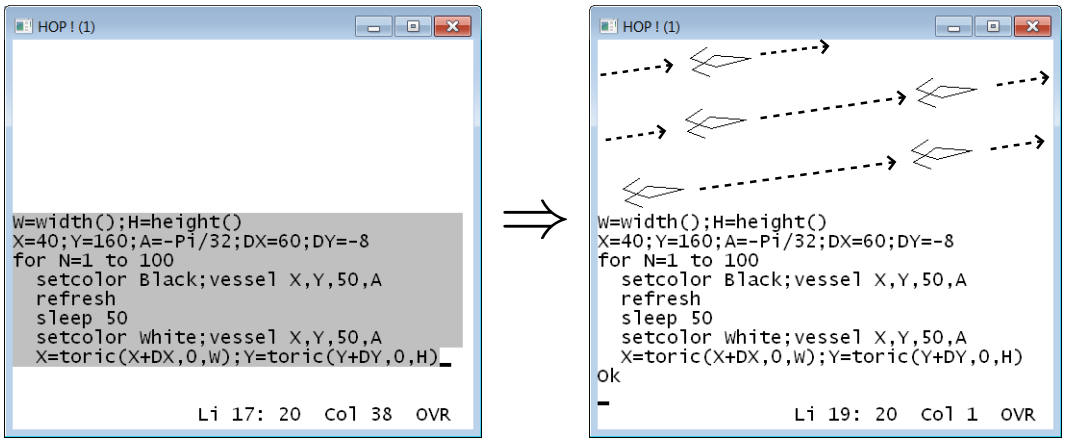
And it is then redrawn a little further (by changing the value of the first parameter of the `vessel()` function):



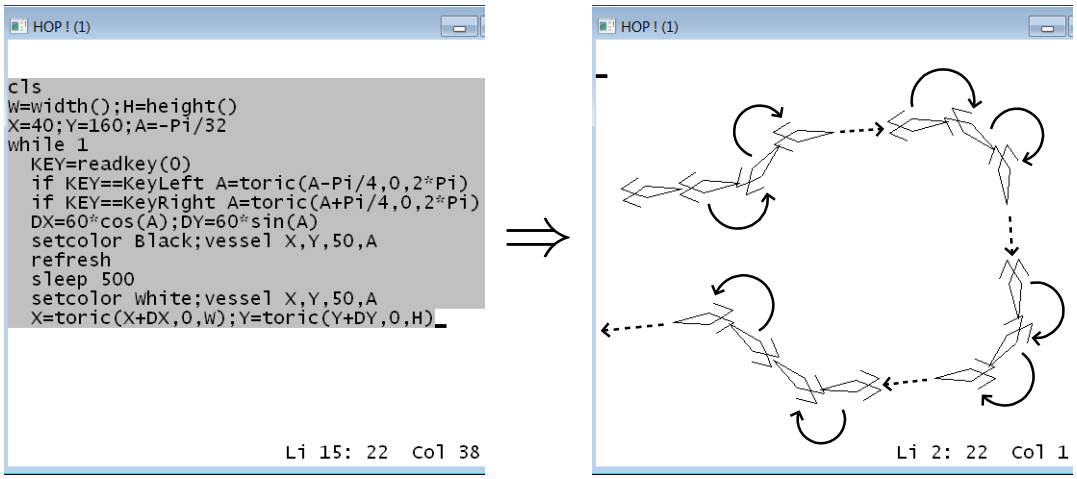
Using the approach outlined above, the animation itself can then be performed by means of a simple loop (note here the use of the `sleep()` and `refresh()` instructions that are needed to implement the animation, which are respectively suspending the program for a given amount of time, and refreshing the screen):



This loop can then be improved to obtain a displacement in a toric space (using the `toric()` function, which calculates the nearest number belonging to a given interval modulo its width):



Finally, the interactive animation of the vessel can be implemented by means of the loop below (note here the use of the `readkey()` instruction, which performs reading from the keyboard):



Overall, of the essential points E3 to E9 mentioned in **4.1**, the following points have effectively been addressed:

- The programming language is small, and comprehensive enough to enable a convenient implementation of the proposed activities (point E3). The environment itself is however not yet built by means of the programming language that learners use ;
- The user interface can actually be used to actively promote the learning of the incremental "hands-on" type approach that we propose (point E4). This user interface could be improved, especially, a certain number of elements related to the functioning of the underlying operating system could be better integrated in the environment, and furthermore, one could seek to provide means to go back to a previous state of the system (see **4.3** below) ;
- The environment provides tools to effectively support the modular approach (point E5). In particular, we use indentation to format blocks, as in Python (as recommended in [GRA06]), and the programming language has a very simple to use system of modules. We have not yet integrated version control tools ;

- The environment has an integrated debugger (point E7), which clearly shows the functioning of the underlying virtual machine[196], but could be further developed ;

Following the preliminary experience we have gained with the system so far, we feel that a number of elements of the HOP! environment should be developed more thoroughly (for more information on this, see **4.3** below).

Finally, the point E6 (look for ways to add language features enabling program verification), the point E8 (achieve an implementation of the HOP! system which is as transparent as possible) and the point E9 (support some of the elements related to didactic and pedagogy to guide the learner) have not been addressed yet.

4.3 Our experience with the HOP! system

A programming course for a period of two months using the HOP! method was proposed and implemented on two occasions at Saarland University (UdS[197]) between August and November 2015, with the participation of about twenty students[198]. The following activities have been proposed[199]:

- A 4-hour initiation activity based on a series of graphical exercises of increasing difficulty using the Logo turtle technique ;
- A reconstruction activity of three video games (Tetris, Asteroids and Pacman), and of a small text editor in the style of Notepad ; in any of these programs, the number of lines of code that learners have to write does not exceed 300 lines (60 lines for Tetris, 200 lines for Asteroids, 300 lines for Pacman, and 90 lines for the editor) ; each one of these programs can be presented and developed by the students in two sessions of four hours each ;

Following is an informal account of the experience gained and of the problems encountered:

- first, there are students interested in the HOP! approach. Who are they ?
  - → *firstly*, science students (students in "hard" sciences, such as physics, but also students in humanities and social sciences where one is dealing with computers, such as psychology, linguistics, or even sociology) wishing to acquire real skills in programming, which is often presented to them informally, or as an "easy" subject: these very students indeed benefit from the real facilities that stem from their mastery of the formal aspects of mathematical or statistical practice, for example, but for them, the result is still that in the absence of an actual practice of software engineering, programs written quickly to meet an immediate need (which can end up lasting much longer than had been planned originally) often turn out to be difficult to maintain and evolve thereafter ;
  - → *secondly*, students who have not done scientific studies, who are interested in computers, or even who are faced with the need to write programs as part of their activity (e.g. language teachers wishing to carry out questionnaires, or write programs to manipulate data) ;

196. See http://hop-project.org/HOPTutorial.pdf

197. Universität des Saarlandes (Germany) ; see http://www.uni-saarland.de

198. We would like to thank the AStA (i.e. the "Allgemeine Studierenden Ausschuss" of the students of the UdS) for its assistance, particularly for providing us the room where we conducted our teachings based on the HOP! method.

199. See http://www.hop-project.org/lectures.html

→ *thirdly*, informatics students who are discouraged, or even frightened by the too-abstract[200] (or conversely, too vocational[201]) approach that they are made to follow in their learning, especially because it does not provides clear answers to certain of their questions, for example about the more or less real skills obsolescence that many people talk about[202], or about the discriminatory practices prevalent in the industry (especially: gender[203] and age[204] related discrimination), which are all things they fear being faced with one day or another ;

- next, not all students have the know-how that the mastery of analytical reasoning requires. Especially:

  → when a proper understanding of the metaphor through which the operating system provides access to the informations contained in the computer is lacking (e.g. when the learner lacks a good command of the handling of files using the file explorer or the shell, or even when she doesn't have a good command of the use of a text editor, or of how it creates default extensions), it becomes difficult for her to perform the basic actions that are necessary for practicing structured programming, such as creating or installing a `HOP!` module (which is a text file located somewhere on the disk, in the folder dedicated to the program one is developing), or even installing or configuring the `HOP!` environment itself (this sometimes requires editing configuration files ; the concept of default path is also something one needs to know).

200. « In many places, there is a disconnect between computer science education and what industry needs. Consider the following exchange:
Famous CS professor (proudly): *"We don't teach programming; we teach computer science."*
Industrial manager: *"They can't program their way out of a paper bag."*
[...]
So what can we do? Industry would prefer to hire "developers" fully trained in the latest tools and techniques whereas academia's greatest ambition is to produce more and better professors. To make progress, these ideals must become better aligned. » [STR10]

201. « I don't think universities should be in the business of teaching things that you should learn at a trade school; I don't think it is the role of a university to teach people how to use, let's say, Visual C++ and its Integrated Development Environment. [...] universities should be teaching things which are likely to last, for a lifetime if you're lucky, but at least 5 or 10 or 20 years, and that means principles and ideas. At the same time, they should be illustrating them with the best possible examples taken from current practice. » [KER00]

202. « There is no really revolutionary technology in aerospace, not in the sense that there aren't any new ideas, but that for more than 30 years, there is no new technology that has radically changed the way we accomplish goals in this field. I would like to speculate that the situation is now similar in computer science and computing technology. From reading press releases, you would think that we experience true revolutions every day. I used to read profiles of startup firms in the newspaper, and invariably, the description of the activities of the startups included the phrase: "X has developed a technology to Y." As far as I could tell, what they did was to write a program, which is hardly a new technology or a true revolution. One gets the impression that before the mid 1990s, nobody wrote any programs, or at least that they wrote only simple low-tech programs. » [BAR05]

203. « Athena Factor survey data show that 41% of highly qualified scientists, engineers, and technologists on the lower rungs of corporate career ladders are female. [...] [But] Over time, fully 52% of highly qualified females working for science, engineering, and technology (SET) companies quit their jobs, driven out by hostile work environments and extreme job pressures. [...] The Athena Factor survey data describe a workplace culture that is at best unsupportive and at worst downright hostile to women. [...] A surprising number of SET women described their male colleagues as acting in rude, predatory, and vulgar ways. [...] Another sobering finding is the degree to which women at SET companies believe that their male colleagues consider females to be intrinsically less capable. In engineering and technology, more than a quarter of female respondents feel they are seen as genetically disadvantaged in SET fields. When Larry Summers, then-president of Harvard University, talked about how women have a "different availability of aptitude" in his infamous speech of January 2005, he was perhaps merely giving voice to a standard view! » [HEW08]

204. « Well, say you interview as a graduating college senior at Facebook Inc. You may find, to your initial delight, that the place looks just like a fun-loving dorm – and the adults seem to be missing. But that is a sign of how the profession has devolved in recent years to one lacking in longevity. Many programmers find that their employability starts to decline at about age 35. Employers dismiss them as either lacking in up-to-date technical skills – such as the latest programming-language fad – or "not suitable for entry level." In other words, either underqualified or overqualified. That doesn't leave much, does it? Statistics show that most software developers are out of the field by age 40. [...] Why do the employers prefer to hire the new or recent grads? Is it really because only they have the latest skill sets? That argument doesn't jibe with the fact that young ones learned those modern skills from old guys like me. Instead, the problem is that the 35-year-old programmer has simply priced herself out of the market. As [noted elsewhere], even if the 45-year-old programmer making $120,000 has the right skills, "companies would rather hire the younger workers." Whether the employers' policy is proper or not, this is the problem facing workers in the software profession. » [MAT12]

We partially solved these problems by simplifying to the extreme the installation process of the `HOP!` system (one just has to copy/paste a folder) and by preserving the system state from one session to another, but it may be appropriate to further integrate such aspects, so as to hide them completely. Furthermore, taking the time to teach these things explicitly when necessary is probably unavoidable ;

→ when the learner is not at all familiar with some of the analytical aspects that programming requires (for example, the idea that one can replace concrete values by the names that *represent* them, to reason abstractly about them), providing the steps to take to build the software is not enough, even when the learner actually manages to obtain the required results: *real* understanding is then lacking. In other words: the "hands-on" approach can only function *from the moment* when a certain number of prerequisites have been assimilated.

Here again, taking the time to explicitly teach this sort of things when necessary is probably essential. To do this, we may have to teach some math, and then start with a number of algorithmic exercises on paper. Moreover, further developing the role of types and introducing reasoning tools that can operate on the program under development in the `HOP!` environment could enable learners to become more easily familiar with the analytical approach which they lack. The question here is firstly to know to what extent this approach can be learnt interactively by means of a computer (vs. "on paper" activities) ; what are the advantages and disadvantages of both media ?

Secondly, how to make analytical reasoning *interesting*, and not be forced to present it as a barren but essential prerequisite ? How to combine *from the start* the learning of analytical thinking with creative activities ? Our experience thereon indicates that graphical programming activities in the style of Logo make it difficult to do this, because these activities are too complex for the students to be really able to directly rediscover by themselves the analytical elements pertaining to the inner working of the programming language: one needs a sizeable dose of *explicit* storytelling, before reaching a stage where the student practices while understanding well enough what she is doing to be able to intelligently observe what she sees and starts to effectively assimilate the underlying mechanisms, their consequences, and how to think about them. Here, it seems to us that one should look for a *simpler* kind of activity, which would be more *directly* aimed at making understand the analytical aspects that one wishes to make the learner assimilate. But it should however always be a *thematic* activity, which does not boil down to a pure manipulation of abstract symbols devoid of a concrete referent that makes sense to the student ;

- finally, we observed that it is not always easy for students to make the link properly between the manipulations that are shown in the tutorial and what actually happens during the interactive activity that stems from their interpretation of the text of this tutorial. Furthermore, during this same interactive activity, operating errors are all the more frustrating that when they happen, one loses all or part of the system state that was built as a result of the sequence of the preceding steps of the interaction.

  It seems to us that here, one should look for ways to integrate the tutorial in the programming environment (so that for example, there are signals that indicate to the learner that she goes in the right direction, that she has completed this or that step, etc.), as well as to provide means to go back to a previous state of the system and to the stage of interaction associated with it (one finds a similar proposal in [ROS99][205]) ;

205. « The undo feature, which currently allows undoing source code changes only, could be extended to undo changes to the run-time state of the program or even side-effects to the environment [...]. » [ROS99]

Overall, of the essential points 1 to 10 mentioned in **4.1**, the following points have effectively been addressed:

- The proposed activity actually draws on a pedagogy based on *reconstructing* software, which draws on *examples* (point 1 and point 2) ;
- The proposed programs have been reconstructed by the learners following an incremental approach that proceeds in stages ; the learners were guided in their work by the author of this article, and by the instructions given in a detailed tutorial outlining the sequence of operations required to reconstruct the software (point 4 and point 9) ;

Following the observations which have been discussed above, we feel that the following elements of the HOP! system should be developed more thoroughly:

- students need *a real practice of software engineering* (point 5), which to be developed, requires the design of software reconstruction activities that are larger than the small examples we have proposed so far ; by moving in this direction, we will be able to better illustrate the big ideas of the domain and the state of the art, as well as to provide activities where more complex *diagnostic tasks* (point 7) may be planned and implemented by students (for now, the reconstruction process being fully spelled out in the tutorials, the students met few bugs, but however, they made mistakes that gave us the opportunity to make them practice debugging) ;
- from the standpoint of understanding the metaphor through which the operating system provides access to information, as well as from the standpoint of the basic analytical aspects that programming requires, *not all students have the know-how that the mastery of analytical reasoning requires*: here, explicit teaching activities related to the point 3 (having a good mental model of the underlying abstract machine[206]) seem to deserve to be developed ; furthermore, a certain number of elements related to the functioning of the underlying operating system could probably be better integrated in the HOP! environment itself ; Finally, explicit teaching of reasoning seems to be required (points 5 and 6), by working "on paper", or possibly by means of computer tools, the exact form of which remaining to be defined ;
- *it is not always easy for students to make the link properly between the manipulations that are shown in the tutorial and what actually happens during the interactive activity*: here, we need better ways to guide the programmer (point 9) ; looking for ways to integrate the tutorial in the programming environment seems a promising direction of development in this regard. Furthermore, the environment currently lacks means to go back to a previous state of the system and to the stage of interaction associated with it (point 4) ;

Finally, the point 8 (learn how to grasp multiple levels of abstraction simultaneously) and the point 10 (make the students work on projects that give them the opportunity to practice writing software specifications) of the HOP! method have not been addressed yet.

206. This includes the programming language, along with the metaphors that are implemented by the operating system.

5 Conclusion

After the analysis conducted in this paper, we are now convinced that it is essential to develop a teaching of scientific and technical subjects that is truly based on discovery and on the practice of plausible reasoning. Still today, the approach to the practice of mathematics described by Polya [POL54] is to our knowledge not used in schools, and little used in higher education.

In the case of informatics, we have seen that the disappointing philistinism which is prevalent in the computer industry, as well as the lack of a well-established tradition on which universities could rely [BAR05][207] mean that one tends to move towards the lowest common denominator, i.e. towards an education that is either too remote from software development practices as they are conducted in reality [DEW08a][208][STR08][209], or too purely vocational[210]. The result is that we lose sight of the concepts, and enter a vicious circle, where the quality of the products made by the industry is in decline, and where universities and schools are unable to redress the situation [STR10][211].

In such a context, where the sustainability of the intellectual ecosystem of the field itself seems in jeopardy, it appears to us that it is urgent to act, and to try to concretely show how one could develop a teaching method which would be both better, because it would try to teach *all* the cognitive aspects that pertain to the problem-solving activity in informatics, and more democratic, the system and teaching materials being distributed under a free license on the one hand, and a real implementation on the ground being undertaken on the other hand, which must be subject to reporting, the reports being also freely accessible.

In this paper, we have laid the foundations of the `HOP!` system's philosophy, and reported on the preliminary experience we have gained with this system. The first version of the `HOP!` environment has been implemented, and a programming course for a period of two months using the `HOP!` method has been developed and implemented.

Following this experience, we plan to develop the project further by following the proposals set out in **4.3** on the one hand, and to move more frankly towards a system dedicated to *discovery-based education* (DbE) on the other hand, by seeking rigorous evaluation methods and criteria that might suit this educational approach. This would give us the means to adopt a less prescriptive approach which would enable us to follow the learner in a more individualized way, to guide her effectively in her approach of discovering the domain, as well as in her discovery of the method that best suits her personal intellectual development, along with the maturation of her own style.

---

207. « I would like to [...] emphasize this disparity between what we teach in CS and what is taught in the mature disciplines of science and engineering. [...] The comparison with physics is also important from the point of view of the *legitimacy* of the curriculum and the teacher. [...] Physics educators have achieved a legitimacy that enables them to dictate a learning sequence that is not affected by trends and fashions.
I believe that CS education must fundamentally change in order to equip the student with a firm, deep and broad theoretical background, long before specialization is undertaken. We are grown up now and with growing up comes the responsibility to build a mature system of education. » [BAR05]

208. « Part of the trouble with universities is that there is relatively few faculty members who know much about software. They know about the theory of computer science and the theory of programming languages. But there are relatively few faculty who really are programmers and software engineers and understand what's involved in writing big applications. » [DEW08a]

209. « That is, programming was seen as a lowly skill that students either did not need or could easily pick up on their own. I have seen the result of that attitude in new graduate students: It is rare that anyone thinks about the structure of their code or the implications for scaling and maintenance – *those are not academic subjects*. » [STR08]

210. « A project leader at an aerospace company once told me that in his experience it is easier to teach computing to a physics major than it is to teach physics to a computer science major. Given the firm theoretical foundation of students of even a concrete subject like mechanical engineering compared with the *artifact-laden* education of a computer science student, this statement is not at all surprising. » [BAR05]

211. « It is essential that there is a balance between the theoretical and the practical—CS is not just principles and theorems, and it is not just hacking code. [...] Many professors will object: "I don't have the time to program!" However, I think that professors who teach students who want to become software professionals will have to make time and their institutions must find ways to reward them for programming. The ultimate goal of CS is to help produce better systems. Would you trust someone who had not seen a patient for years to teach surgery? What would you think of a piano teacher who never touched the keyboard? » [STR10]

Appendix A Bibliography

[ADE85] B. Adelson and E. Soloway, *The role of domain experience in software design*, IEEE Transactions on software engineering, 11, pp. 1351-60, 1985.

[AGI01a] Beck et al., *Twelve Principles of Agile Software*, 2001. `http://www.agilemanifesto.org/principles.html`

[AGI01b] Beck et al., *Manifesto for Agile Software Development*, 2001. `http://www.agilemanifesto.org/iso/en`

[AND82] John R. Anderson, *Acquisition of Cognitive Skill*, Psychological Review, Vol. 89, pp. 369–406, 1982.

[AND93] John R. Anderson, *Rules of the Mind*, Hillsdale, NJ: Erlbaum, 1993.

[ARE58] Hannah Arendt, *The crisis in Education*, in *Between Past and Future: Eight Exercises in Political Thought*, New York, Viking Press, 1961.

[AST95] Owen Astrachan and David Reed, *The applied apprenticeship approach to CS 1*, 26th SIGCSE Technical Symposium on Computer Science Education, pp. 1-5, 1995.

[BAC07] R. J. Back, J. Eriksson and L. Mannila, *Teaching the Construction of Correct Programs Using Invariant Based Programming*, In Proceedings of the 3rd South-East European Workshop on Formal Methods, 2007.

[BAR05] Mordechai Ben-Ari, *The Concorde Doesn't Fly Anymore*, Keynote Talk, SIGCSE 2005.

[BAR64] Paul Baran, *On Distributed Communications*, Memorandum RM-3420-PR, Rand Corporation, 1964.

[BAS75] V. R. Basili and A. J. Turner, *Iterative Enhancement: A Practical Technique for Software Development*, IEEE Trans. Software Engineering, Vol. 1, No. 4, pp. 390-396, 1975.

[BBC14] *Tech giants spend millions to stop another Heartbleed*, BBC Technology, April 2014. `http://www.bbc.com/news/technology-27155946`

[BER89] Tim Berners-Lee, *Information Management: A Proposal*, CERN Memo, March 1989, May 1990.

[BER91] Tim Berners-Lee, *HyperText Transfer protocol (HTTP) 0.9*, World Wide Web Consortium, 1991.

[BLA02] A. Blackwell, *What is Programming?*, PPIG, Brunel University, London, UK, 2002.

[BLO85] Benjamin Bloom (ed.), *Developing Talent in Young People*, Ballantine, 1985.

[BOE00] Barry Boehm and Victor R. Basili, *Gaining Intellectual Control of Software Development*, IEEE Computer, 2000.

[BOE10] Barry Boehm, *Architecting: How Much and When?*, in *Making Software: What Really Works, and Why We Believe It*, Andy Oram and Greg Wilson (Eds.), pp. 161-185, O'Reilly 2010.

[BOU66] Pierre Bourdieu, *L'école conservatrice. Les inégalités devant l'école et devant la culture*, Revue française de sociologie, Vol. 7, No 3, pp. 325-347, 1966.

[BRA99] John D. Bransford, Ann L. Brown, and Rodney R. Cocking (Eds.), *How People Learn: Brain, Mind, Experience, and School*, Commission on Behavioral and Social Sciences and Education, National Research Council, National Academy Press, Washington, D.C., 1999.

[BRO14] Jon Brodkin, *Tech giants, chastened by Heartbleed, finally agree to fund OpenSSL*, Ars Technica, 2014. `http://arstechnica.com/information-technology/2014/04/tech-giants-chastened-by-heartbleed-finally-agree-to-fund-openssl/`

[BRO75] Frederick P. Brooks, *The mythical man-month*, Addison Wesley, 1975.

[BRO86] Frederick P. Brooks, *No Silver Bullet – Essence and Accidents of Software Engineering*, Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-76, 1986.

[BRI06] David Brin, *Why Johnny can't code*, Salon.com, 2006. `http://www.salon.com/2006/09/14/basic_2/`

[CAN82] Howard I. Cannon, *Flavors: A non-hierarchical approach to object-oriented programming*, Symbolics Inc., 1982.

[CAR99] Henri Cartan and Allyn Jackson, *Interview with Henri Cartan*, Notices of the AMS, Vol. 46, No. 7, pp. 782-788, 1999. `http://www.ams.org/notices/199907/fea-cartan.pdf`

[CER74] Vinton Cerf et al., *RFC 675 – Specification of Internet Transmission Control Protocol*, 1974.

[CHA09] Scott Chacon and Ben Straub, *Pro Git*, Apress, 2009.

[CHI81] M. T. H. Chi, P. J. Feltovich and R. Glaser, *Categorization and Representation of Physics Problems by Experts and Novices*, Cognitive Science, 5, pp. 121-152, 1981.

[COD70] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, Vol. 13, No. 6, pp. 377-387, 1970.

[COM84] *Commodore 64 User's Guide*, Commodore Business Machines, 1984.

[CON93] Charles Consel and Olivier Danvy, *Tutorial Notes on Partial Evaluation*, Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, pp. 493–501, 1993.

[COO07] William Cook, *AppleScript*, History of programming languages (HOPL III). Proceedings of the third ACM SIGPLAN conference (ACM):1–21, 2007.

[COR69] F. J. Corbató, *PL/I As a Tool for System Programming*, May 1969.

[CRE81] R.J. Creasy, *The origin of the VM/370 time-sharing system*, IBM Journal of Research & Development, Vol. 25, No. 5, pp. 483–90, 1981.

[CUS95] Michael A. Cusumano and Stanley Smith, *Beyond the Waterfall: Software Development at Microsoft*, MIT Sloan School of Management, Working Paper #3844-BPS-95, 1995.

[DAH66] Ole-Johan Dahl and Kristen Nygaard, *SIMULA: an ALGOL-based simulation language*, Communications of the ACM, Vol. 9, No. 9, pp. 671-678, 1966.

[DAH67] Ole-Johan Dahl and Kristen Nygaard, *Class and subclass declarations*, In Proceedings from IFIP TC2 Conference on Simulation Programming Languages, Lysebu, Oslo, ed.: J. N. Buxton, pp. 158-174. North Holland, 1967.

[DAL65] R.C. Daley, P.G. Neumann, *A General-Purpose File System For Secondary Storage*, AF/PS Conference Proceedings, Vol. 27, Fall Joint Computer Conference, 1965.

[DAV12] Jim Davies and David Matheson, *The cognitive importance of testimony*, Principia 16(2): 297–318, Published by NEL—Epistemology and Logic Research Group, Federal University of Santa Catarina (UFSC), Brazil, 2012.

[DEG65] A.D. De Groot, *Thought and Choice in Chess*, The Hague, the Netherlands: Mouton, 1965.

[DES29] René Descartes, *Rules for the Direction of the Mind* (1629), in Elisabeth Anscombe and Peter Thomas Geach, *Descartes: Philosophical Writings*, Edimburgh, Thomas Nelson and Sons, Ltd., 1954. `https://en.wikisource.org/wiki/Rules_for_the_Direction_of_the_Mind`

[DEW38] John Dewey, *Experience and Education*. New York: Macmillan, 1938.

[DEW08a] James Maguire, *The 'Anti-Java' Professor and the Jobless Programmers*, IT Business Edge, QuinStreet Eds., 2008.

[DEW08b] Robert B. K. Dewar and Edmond Schonberg, Computer Science Education: *Where Are the Software Engineers of Tomorrow?*, The Journal of Defense Software Engineering, pp. 28-30, 2008.

[DIJ68] Edsger W. Dijkstra, *The structure of the THE multiprogramming system*, Comm. ACM, Vol. 11, pp. 341-346, 1968.

[DIJ68b] Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, Vol. 11, No. 3, pp. 147-148, 1968.

[DIJ69] Edsger W. Dijkstra, *Notes on Structured Programming* (EWD 249), 1969.

[DIJ71] Edsger W. Dijkstra, *On the Reliability of Programs*, E. W. Dijkstra Archive (EWD303), 1971. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF`

[DIJ75] Edsger W. Dijkstra, *How do we tell truths that might hurt?* (EWD498), 1975.

[DIJ89] Edsger W. Dijkstra, *On the Cruelty of Really Teaching Computing Science*, Comm. ACM, Vol. 32, pp. 1398-1404, 1989. `http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF`

[DOW13] Gilles Dowek, *Une démonstration est-elle une explication ?*, La preuve et ses moyens, Journées de Rochebrune 2013, 2013.

[DUB89] B. du Boulay, *Some difficulties of learning to program*, In E. Soloway & J.C. Spohrer (Eds.), pp. 283–299, Hillsdale, NJ: Lawrence Erlbaum, 1989.

[EHR01] L.C. Ehri, S.R. Nunes, D.M. Willows, B.V. Schuster, Z. Yaghoub-Zadeh and T. Shanahan, *Phonemic awareness instruction helps children learn to read : evidence from the National reading Panel's meta-analysis*, Reading Research Quarterly, 36(3), 250-287, 2001.

[ENG62] Douglas C. Engelbart, *Augmenting Human Intellect: A Conceptual Framework*, SRI Summary Report AFOSR-3223, Washington DC, October 1962.

[ERA11] Erasmus, *The Praise of Folly* (1511), translated by John Wilson, 1668. http://www.gutenberg.org/ebooks/9371

[FEY99] Richard Feynman, *The Pleasure of Finding Things Out*, Perseus Books, New York, pp. 186-187, 1999.

[FRE06] T. Freeth, Y. Bitsakis, X. Moussas, J.H. Seiradakis et al., *Decoding the ancient Greek astronomical calculator known as the Antikythera Mechanism*, Nature 444 (7119): pp. 587–591, 2006.

[FLS04] *Flying Saucer*, 2004. https://code.google.com/archive/p/flying-saucer/

[GAR95] David Garlan, Robert Allen and John Ockerbloom, *Architectural Mismatch or Why it's hard to build systems out of existing parts*, ICSE'95, Seattle, Washington USA, 1995.

[GEA07] David C. Geary, *Educating the Evolved Mind: Conceptual Foundations for an Evolutionary Educational Psychology*, in Psychological Perspectives on Contemporary Educational Issues, edited by J. S. Carlson & J. R. Levin. Greenwich, CT: Information Age Publishing, 2007.

[GOL89] Adele Goldberg, David Robson, *Smalltalk-80 The Language*. Addison Wesley, 1989. ISBN 0-201-13688-0.

[GRA01] Paul Graham, *Beating the Averages* (2001), in *Hackers & Painters: Big Ideas from the Computer Age*, O'Reilly Media, 2004.

[GRA06] Linda Grandell, Mia Peltomäki, Ralph-Johan Back and Tapio Salakoski, *Why Complicate Things? Introducing Programming in High School Using Python*, Eighth Australasian Computing Education Conference (ACE2006), 2006.

[GRE04] J. Greenfield, K. Short, S. Cook and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[GRI12] Avdi Grimm, *Let's stop telling programming newbies to learn Vim (or Emacs)*, Virtuous Code, 2012. http://devblog.avdi.org/2012/10/16/lets-stop-telling-program ming-newbies-to-learn-vim-or-emacs

[GRI75] R.E. Griswold, *Extensible pattern matching in SNOBOL4*. In proceedings of the ACM Annual Conference, pp. 248-252. New York: Association for Computing Machinery, 1975.

[GUI06] Nicolas Guibert, *Validation d'une approche basée sur l'exemple pour l'initiation à la programmation*, Thèse pour l'obtention du grade de Docteur de l'Université de Poitiers, 2006.

[GWB86] *Microsoft GW-BASIC User's Guide and User's Reference*, Microsoft, 1986.

[HAG12] James Hague, *A Complete Understanding is No Longer Possible*, http://prog21.dad gum.com/129.html, 2012.

[HAG15] James Hague, *If You Haven't Done It Before, All Bets Are Off*, http://prog21.dad gum.com/209.html, 2015.

[HAN75] P. Brinch Hansen, *The programming language Concurrent Pascal*, IEEE Transactions on Software Engineering 1, 2, pp. 199-207, 1975.

[HAR63] Timothy P. Hart, *MACRO Definitions for LISP*, AI memo #57, RLE and MIT Computation Center, 1963.

[HAY77] J.R. Hayes and H.A. Simon, *Psychological differences among problem isomorphs*, In Cognitive Theory (eds. Castellan N. J., Jr., Pisoni D B. & Potts G. R.), Vol. 2. Erlbaum, Hillsdale, NJ, 1977.

[HAY89] John R. Hayes, *Complete Problem Solver*, Lawrence Erlbaum, 1989.

[HEN76] Peter Henderson and James H. Morris, *A Lazy Evaluator*, Conference Record of the 3rd ACM symposium on Principles of Programming Languages, 1976.

[HEN04] Philip H. Henning, *Everyday Cognition and Situated Learning*, In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology (2nd ed.)*, pp. 143-168, Mahwah, New Jersey: Lawrence Erlbaum Associates, 2004.

[HER07] Esther Herrmann, Josep Call, María Victoria Hernández-Lloreda, Brian Hare, Michael Tomasello, *Humans Have Evolved Specialized Skills of Social Cognition: The Cultural Intelligence Hypothesis*, Science, September 2007.

[HEW08] Sylvia Ann Hewlett, Carolyn Buck Luce, Lisa J. Servon, Laura Sherbin, Peggy Shiller, Eytan Sosnovich, and Karen Sumberg, *The Athena Factor: Reversing the Brain Drain in Science, Engineering, and Technology*, Harvard Business Review, Research Report #10094, 2008.

[HIP15a] Andrew W. Torrance and Eric von Hippel, *The Right to Innovate*, Michigan State Law Review 2015:793, pp. 793-829, 2015. http://ssrn.com/abstract=2339132

[HIP15b] Alfonso Gambardella, Christina Raasch and Eric von Hippel, *The user innovation paradigm: impacts on markets and welfare*, 2015. Available at SSRN: http://ssrn.com/abstract=2079763 or http://dx.doi.org/10.2139/ssrn.2079763

[HOA69] C.A.R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM 12(10):576–580, 1969.

[HOA74] C.A.R. Hoare, *Monitors: an operating system structuring concept*, Communications of the ACM 17(10):549–557, 1974.

[HOF95] Douglas Hofstadter, *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*, Basic Books, 2008. ISBN 0786723319

[HOU13] Cory House, *Clean Code Matters*, http://blog.pluralsight.com/7-reasons-clean-code-matters, 2013.

[HUA11] Shan Shan Huang, Todd Jeffrey Green and Boon Thau Loo, *Datalog and emerging applications: an interactive tutorial*, Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 1213-1216, 2011.

[HUG90] John Hughes, *Why Functional Programming Matters*, in Research Topics in Functional Programming, Addison-Wesley, pp. 17–42, 1990.

[IRO72] Edgar T. Irons, Frans M. Djorup, *A CRT editing system*, Communications of the ACM, 15(1):16–20, 1972.

[JEN02] Tony Jenkins, *On the Difficulty of Learning to Program*, in *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, Loughborough, UK, 2002. http://www.psy.gla.ac.uk/~steve/localed/jenkins.html

[JOE01] Joel Spolsky, *Strategy Letter IV: Bloatware and the 80/20 Myth*, Joel on Software, 2001.

[JOH78] S. C. Johnson, D. M. Ritchie, *Portability of C Programs and the UNIX System*, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 2021-2048, 1978.

[KAY12] Andrew Binstock and Alan Kay, *Interview with Alan Kay*, Dr. Dobb's Journal, 2012.

[KEE88] Sonya Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1988.

[KEM68] John G. Kemeny and Thomas E. Kurtz, *BASIC, Dartmouth College Computation Center*, 1968.

[KER00] Brian Kernighan, Mihai Budiu, *An Interview with Brian Kernighan*, http://www.cs.cmu.edu/~mihaib/kernighan-interview, 2000.

[KET12] Rafe Kettler, *A Guide to Python's Magic Methods*, 2012. http://www.rafekettler.com/magicmethods.pdf

[KIC91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[KIC97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect-oriented programming*, Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), LNCS 1241, pp. 220–242, 1997.

[KIN76] J. C. King, *Symbolic execution and program testing*, Comm. ACM 19, pp. 385-394, 1976.

[KIR06] Paul A. Kirschner , John Sweller and Richard E. Clark, *Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*, Educational Psychologist, 41:2, 75-86, 2006.

[KNU74] Donald Knuth, *Computer Programming as an Art (1974 ACM Turing Award Lecture)*, CACM, Vol. 17, No. 12, 1974.

[KNU84] Donald Knuth, *The TEX book*, Addison-Wesley Publishing Company, 1984.

[KNU99] Donald E., Knuth, *MMIXware: A RISC Computer for the Third Millennium*, Lecture Notes in Computer Science Tutorial 1750, 1999.

[KNU02] Donald Knuth, *Donald E. Knuth über MMIX und die Kunst des Programmierens*, http://heise.de/-288138, 2002.

[KNU03] Donald E. Knuth, *Bottom-up education*, ITiCSE '03 Proceedings of the 8th annual conference on Innovation and technology in computer science education, 2003.

[KOL05] Y. Ben-David Kolikant, *Students' alternative standards for correctness*, In Proceedings of the First International Workshop on Computing Education Research, pages 37-43. ACM Press, 2005.

[LAK80] George Lakoff and Mark Johnson, *Metaphors we live by*, University of Chicago Press, 1980.

[LAM86] Leslie Lamport, *LaTeX : A Document Preparation System*, Addison-Wesley Publishing Company, 1986.

[LAS12] Matthew Lasar, *25 years of HyperCard – the missing link to the Web*, Ars Technica, 2012.

[LAU14] Michael B. McLaughlin, Herb Sutter, Jason Zink, *A Proposal to Add 2D Graphics Rendering and Display to C++*, 2014. `https://isocpp.org/blog/2014/05/n4021`

[LEV98] Ralph Levien, *The decommoditization of protocols*, `http://www.levien.com/free/decommoditizing.html`, 1998.

[LEH00] Richard Lehrer and Leona Schauble, *Developing Model-Based Reasoning in Mathematics and Science*, Journal of Applied Developmental Psychology 21(1):39–48, 2000.

[LEH14] Timo O.A. Lehtinen, Mika V. Mäntylä, Jari Vanhanen, Juha Itkonen, Casper Lassenius, *Perceived causes of software project failures – An analysis of their relationships, Information and Software Technology*, No. 56, pp. 623-643, 2014.

[LIS74] B.H. Liskov and S.N. Zilles, *Programming with abstract data types*, Proc. ACM SIGPLAN Symp. on Very High Level Languages, SIGPLAN Notices (ACM) 9, 4, pp. 50-59, 1974.

[LIS11] Raymond Lister, *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*, Proceedings of the Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, 2011.

[LOC13] Information is Beautiful, *Codebases – Millions of lines of code*, `http://bit.ly/KIB_linescode`, 2013.

[MAR00] Robert C. Martin, *Design Principles and Design Patterns*, `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, 2000.

[MAT03] Norman Matloff, *On the Need for Reform of the H-1B Non-immigrant Work Visa in Computer-Related Occupations*, University of Michigan Journal of Law Reform, Fall 2003, Vol. 36, Issue 4, 815-914, 2003. `http://heather.cs.ucdavis.edu/Mich.pdf`

[MAT12] Norman Matloff, *Software Engineers Will Work One Day for English Majors*, Bloomberg View, 2012.

[MCC60] John McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Part I, Communications of the ACM, 3, pp. 184-195, 1960.

[MEE83] L. G. L. T. Meertens, *Incremental Type Checking in B*, Conf. Record of the 9th ACM Symp. on Principles of Programming Languages, 1983.

[MEE91] Lambert Meertens, Steven Pemberton and Guido van Rossum, *The ABC Structure Editor: Structure-based Editing for the ABC Programming Environment*, CWI Report CS-R9256 1992, 1992.

[MER05] Marjan Mernik, Jan Heering and Anthony M. Sloane, *When and How to Develop Domain-Specific Languages*, ACM Computing Surveys, Vol. 37, No. 4, pp. 316–344, 2005.

[MEY97] B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Upper Saddle River, N.J., 1997.

[MEY01] Bertrand Meyer, *Software Engineering in the Academy*, IEEE Computer, 2001.

[MEY03] Jan Meyer and Ray Land, *Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines*, Occasional Report 4, © ETL Project, Universities of Edinburgh, Coventry and Durham, 2003.

[MIL78] R. Milner, *A Theory of Type Polymorphism in Programming*, J. Computer and System Sciences, No. 17, pp. 348-345, 1978.

[MIL89] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[MOJ06] Mac Mojo, *It's all in the numbers*, `http://blogs.msdn.com/b/macmojo/archive/2006/11/03/it-s-all-in-the-numbers.aspx`, 2006.

[MUE02] Matthias Müller-Prove, *Vision and Reality of Hypertext and Graphical User Interfaces*, Master Thesis, Hamburg University, 2002.

[MUE11] J.S. Mueller, S. Melwani, and J. A. Goncalo, *The bias against creativity: Why people desire but reject creative ideas*, Cornell University, 2011.

[MUN14] *Conditions de travail des informaticiens étrangers en France*, MUNCI, 2014. https://munci.org/article1622.html

[NAD05] R. J. Nadolski, P. A. Kirschner and J. J. G. van Merriënboer, *Optimising the number of steps in learning tasks for complex skills*, British Journal of Educational Psychology, 75, pp. 223–237, 2005.

[NAI95] L. Naish, T. Barbour, *Towards a portable lazy functional declarative debugger*, Technical Report 95/23, Departement of Computer Science, University of Melbourne, Australia, 1995.

[NAS96] National Academy of Sciences, *Statistical Software Engineering*, 1996. http://www.nap.edu/html/statsoft/chap2.html

[NAU63] Peter Naur et al., *Revised report on the algorithmic language Algol 60*, Communications of the ACM 6(1):1-17, January 1963.

[NAU69] P. Naur and B. Randell (Eds.), *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*, Brussels, Scientific Affairs Division, NATO, 231pp, 1969.

[NEE72] R.M. Needham, *Protection systems and protection implementations*, AFIPS FJCC, vol. 41, pp, 571-578, 1972.

[NIL98] H. Nilsson, *Declarative Debugging for Lazy Functional Languages*, PhD Thesis, Linköping University, 1998.

[OPE98] *Open SSL*, 1998. https://www.openssl.org/

[PAN00] John Pane, Brad Myers, *The Influence of the Psychology of Programming on a Language Design: Project Status Report*, in Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group, A. F. Blackwell and E. Bilotta (Eds.), pp. 193-205, 2000.

[PAR72] David L. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules*, Comm. ACM, Vol. 15, No. 12, pp. 1053-58, 1972.

[PEA83] R. D. Pea and D. M. Kurland, *On the cognitive prerequisites of learning computer programming*, Tech. Rep. No. 18, New York, NY Bank Street College of Education, Center for Children and Technology. (ERIC Document Reproduction Service No. ED 249 931), 1983.

[PEA87] Roy D. Pea, Elliot Soloway, Jim C. Spohrer, *The Buggy Path to The Development of Programming Expertise*, Focus on Learning Problems in Mathematics, Vol. 9, No. 1, pp. 5-30, 1987.

[PEA11] Alison Pease and Andrew Aberdein, *Five Theories of Reasoning: Inter-connections and Applications to Mathematics*, Logic and Logical Philosophy Vol. 20, No. 1-2, pp. 7-57, 2011.

[PEM14] Steven Pemberton, *The Programming Language as Human Interface*, Chi Sparks 2014 Conference, 2014.

[PEN90] Nancy Pennington and Beatrice Grabowski, *The Tasks of Programming*, in Psychology of Programming, Academic Press, 1990.

[PER86] D. Perkins, R. Hobbs, F. Martin and R. Simmons, *Conditions of Learning in Novice Programmers*, In Studying the Novice Programmer, Lawrence Erlbaum Associates, pp. 261-279, 1986.

[PHO90] The Computer History Museum, *Photoshop version 1.0.1 Source Code*, http://www.computerhistory.org/atchm/adobe-photoshop-source-code

[PIA69] J. Piaget and B. Inhelder, *The Psychology of the Child*, London: Routledge and Kegan Paul, 1969.

[PIA73] Jean Piaget, *To Understand is to Invent*, New York: Grossman, 1973.

[PIK04] Rob Pike, *Rob Pike Responds*, Slashdot, 2004. http://interviews.slashdot.org/story/04/10/18/1153211/rob-pike-responds

[PIL04] C. Michael Pilato, Ben Collins-Sussman and Brian W. Fitzpatrick, *Version Control with Subversion*, O'Reilly, 2004.

[PLATOa] Plato, *The Apology*, 29c-30c, in *The Apology, Phaedo and Crito*, trans. by Benjamin Jowett. Vol. II, Part 1. The Harvard Classics. New York: P.F. Collier & Son, 1909.

[PLATOb] Plato, *The Republic*, 518b, trans. by Allan Bloom, with notes and an interpretive essay. New York: Basic Books, 1968.

[POI05] Henri Poincaré, *Science and Hypothesis*, The Walter Scott Publishing Co., 1905.

[POL45] George Polya, *How To Solve It*, Princeton University Press, 1945.

[POL54] George Polya, *Mathematics and Plausible Reasoning*, Princeton University Press, 1954.

[POP77] K. R. Popper and J. C. Eccles, *The Self and its Brain*, Berlin: Springer-Verlag, 1977.

[POS81] Jonathan B. Postel, *RFC 788 – Simple Mail Transfer Protocol*, 1981.

[RAN70] B. Randell and J.N. Buxton (Eds.), *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969*, Brussels, Scientific Affairs Division, NATO, 164pp, 1970.

[REE79] Trygve Reenskaug, *Thing-Model-View-Editor, an Example from a planning system*, Xerox PARC technical note, May 1979.

[RES09] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai, *Scratch: Programming for All*, Communications of the ACM, Vol. 52, No. 11, pp. 60-67, 2009.

[REU14] Dan Levine, Peter Henderson, *Apple, Google agree to settle lawsuit alleging hiring conspiracy*, Reuters, 2014.

[REY84] J. K. Reynolds, *RFC 918 – Post Office Protocol*, 1984.

[REY99] John C. Reynolds, *Theories of programming languages*, Cambridge University Press, 1999.

[RIT14] Bethany Rittle-Johnson, Michael Schneider, *Developing Conceptual and Procedural Knowledge of Mathematics*, in *The Oxford Handbook of Numerical Cognition*, Oxford University Press, 2014.

[ROB78] Larry Roberts, *The Evolution of Packet Switching*, Invited paper, IEEE, November 1978.

[ROB03] Anthony Robins, Janet Rountree and Nathan Rountree, *Learning and Teaching Programming: A Review and Discussion*, Computer Science Education, Vol. 13, No. 2, pp. 137–172, 2003.

[ROS99] Guido van Rossum, *Computer Programming for Everybody*, Corporation for National Research Initiatives, CNRI Proposal #90120-1a, 1999.

[ROT81] Gian-Carlo Rota, in preface to Philip J. Davis and Reuben Hersh, *The Mathematical Experience*, Birkhäuser, 1981.

[SAB98] Amr Sabry, *What is a purely functional language?*, Journal of Functional Programming Vol. 8, No. 1, pp. 1–22, 1998.

[SAU09] Brian J. Sauser, Richard R. Reilly, Aaron J. Shenhar, *Why projects fail? How contingency theory can provide new insights – A comparative analysis of NASA's Mars Climate Orbiter loss*, International Journal of Project Management 27, pp. 665-679, 2009.

[SCH78] G. Michael Schneider, *The introductory programming course in computer science: ten principles*, In Papers of the 9th SIGCSE/CSA Technical Symposium on Computer Science Education, pp. 107–114. ACM Press, 1978.

[SCH86] Jacob T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming With Sets: An Introduction to SETL*, 1986. ISBN 0-387-96399-5.

[SCH08] Edmond Schonberg and Robert Dewar, *A principled approach to software Engineering Education, or Java considered Harmful*, Ada User Journal, Volume 29, Number 3, 2008.

[SHA82] E. Shapiro, *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1982.

[SHI77] R. M. Shiffrin, W. Schneider, *Controlled and automatic human information processing: II. Perceptual learning, automatic attending and a general theory*. Psychological Review, 84, 127–190, 1977.

[SLE88] D. Sleeman, *An introductory Pascal class: A study of student errors*, In Teaching and learning Computer Programming. R. E. Mayer, Lawrence Erlbaum Associates, 1988.

[SLO96] S. A. Sloman, *The empirical case for two systems of reasoning*, Psychological Bulletin, 119, 3–22, 1996.

[SOL84] E. Soloway, K. Ehrlich, *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. Se-10, No. 5, 1984.

[SOL85] E. Soloway, *From problems to programs via plans: The content and structure of knowledge for introductory LISP programming*. I. Educ. Comput. Res., 1985.

[STA09] J.R. Star, K. J. Newton, *The nature and development of expert's strategy flexibility for solving equations*, ZDM Mathematics Education, 41, 557–567, 2009.

[STA95] Barry Staw, *Creative Action in Organizations, Ivory Tower Visions and Real Voices*, Cameron Ford, Dennis Gioia (eds.), pp. 161-166, 1995.

[STE07] Ian Stewart, *The Future of Proof*, Prospect, March 2007.

[STO15] Stack Overflow, *Stack Overflow Developer Survey*, `http://stackoverflow.com/research/developer-survey-2015`, 2015.

[STR08] Bjarne Stroustrup, James Maguire, *On Educating Software Developers (interview)*, IT Business Edge, QuinStreet Eds., 2008.

[STR10] Bjarne Stroustrup, *What Should We Teach New Software Developers? Why?*, Communications of the ACM, Vol. 53, No. 1, 2010.

[STR14] Bjarne Stroustrup and Richard Morris, *Programmers With Class (interview)*, Simple Talk, 19 August 2014. `https://www.simple-talk.com/content/print.aspx?article=2038`

[SUT63] Ivan Edward Sutherland, *Sketchpad: A man-machine graphical communication system*, University of Cambridge, 1963. University of Cambridge UCAM-CL-TR-574, 2003.

[SWE85] J. Sweller, G. A. Cooper, *The use of worked examples as a substitute for problem solving in learning algebra*. Cognition and Instruction, 2, 59–89, 1985.

[THO78] Ken Thompson, *UNIX Time-Sharing System: UNIX Implementation*, Bell System Technical Journal, Vol. 57, No. 6, pp. 1931–1946, 1978.

[THO84] Ken Thompson, *Reflections on Trusting Trust*, 1983 Turing Award Lecture, Communications of the ACM 27(8), pp. 761-763, 1984.

[TOM08] Michael Tomasello, *Origins of Human Communication*, MIT Press, 2008.

[TRY09] Trygve Reenskaug and James O. Coplien, *The DCI Architecture: A New Vision of Object-Oriented Programming*, 2009.

[VEL95] T. Veldhuizen, *Using C++ template metaprograms*, C++ Report, Vol. 7, No. 4, pp. 36-43, 1995.

[VES85] I. Vessey, *Expertise in debugging computer programs: A process analysis*, International Journal of Man-Machine Studies, pp. 459-494, Vol. 23, 1985.

[VYG81] Lev S. Vygotsky, *The genesis of higher mental functions*. In J.V. Wertsch (ed.), The Concept of Activity in Soviet Psychology (pp. 144–188). Armonk, NY: Sharpe, 1981.

[WEI80] J. L. Weiner, *Blah: a System Which Explains its Reasoning*, Artificial Intelligence, 15(1-2), pp. 19-48, 1980.

[WEL09] Gordon C. Wells, *The Meaning Makers: Learning to Talk and Talking to Learn*, Multilingual Matters, 2009.

[WEL00] Gordon C. Wells, *Dialogic Inquiry in Education: Building on the Legacy of Vygotsky*, In C.D. Lee and P. Smagorinsky (eds.), *Vygotskian perspectives on literacy research*. New York: Cambridge University Press, (pp. 51-85), 2000.

[WHE01] David A. Wheeler, *More Than a Gigabuck: Estimating GNU/Linux's Size*, 2001. `http://www.dwheeler.com/sloc`

[WHE15] David A. Wheeler, *How to Prevent the next Heartbleed*, 2015. `http://www.dwheeler.com/essays/heartbleed.html`

[WHI13] Kaitlynn M. Whitneya, Charles B. Daniels, *The Root Cause of Failure in Complex IT Projects: Complexity Itself*, Conference on Complex Adaptive Systems, Missouri University of Science and Technology, Cihan H. Dagli, Ed., Procedia Computer Science 20, pp. 325-330, 2013.

[WIC92] M. R. Wick, W. B. Thompson, *Reconstructive expert system explanation*, Artificial Intelligence, 54(1-2), pp. 33-70, 1992.

[WIC95] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward and D. W. R. Marsh, *An industrial perspective on static analysis*, Software Engineering Journal, Vol. 10, No. 2, pp. 69-75, 1995.

[WIR71] Niklaus Wirth, *Program development by stepwise refinment*, Comm. ACM, Vol. 14, pp. 221-227, 1971.

[WIR75] Niklaus Wirth, *An Assessment of the Programming Language Pascal*, IEEE Transactions on Software Engineering, Vol. 1, No. 2, pp. 192-198, 1975.

[WIR80] Niklaus Wirth, *MODULA-2*, ETH Zürich, 1980.

[WIR95] Niklaus Wirth, *A Plea for Lean Software*, IEEE Computer, Vol. 28, No. 2, pp. 64-68, 1995.

[WOR91] The Computer History Museum, *Microsoft Word for Windows Version 1.1a Source Code*, `http://www.computerhistory.org/atchm/microsoft-word-for-windows-1-1a-source-code`

[XIA07] Bi Xiaolei, *L'évolution historique de la pédagogie traditionnelle en Chine*, Le Journal des Chercheurs, 2007. `http://www.barbier-rd.nom.fr/journal/spip.php?article741`

[YAN03] Hongji Yang and Martin Ward, *Successful Evolution of Software Systems*, Artech House, 2003. ISBN 1-58053-349-3.