Developing a programming design taste

Eduardo Bellani

October 7, 2009

Abstract

A latent potential for emergent education in programming exists, but most of the widely available material focus on wasteful details and does not help the learner develop mental techniques to judge what is a beautiful program and how to manage its own thinking patterns. In this article I try to present a way that would address this issue, by providing the tools for a person to become aesthetically proficient in programming, and through that develop administrative thinking patterns.

keywords: education, programming, design, taste

1 Introduction

The problem that I will cover in this article is the one of learning how to think programmatically, that is, to program.

What I will be focusing is specially the demonstration of paths and technique's a single aspirant programmer would use to grasp not only the minor surface details of the craft, but its meaning and consequences, so that it could have a chance at approaching complex and dynamic problems. The same holds true for a teacher who wants to build a class for novice programmers.

There is a latent potential for an increase in the depth of the knowledge about computation in the general population. You can observe for instance it by the appreciation of computer games and graphics. Sure, not every gamer or graphic designer has a desire for becoming a programmer, not even the majority. But one can assume that from the enormous pool of people involved in those activities there could be an significant category interested in computing itself.

Besides this, a lot of benefits comes from learning to program, even if we ignore the monetary incentive. Sussman [2005] Not only that, but this mastery could lead to an increase in critical thinking and multiple levels of abstraction. As Felleisen et al. [2001] states:

> On one hand, program design teaches the same analytical skills as mathematics. But, unlike mathematics, working with programs is an active approach to learning. Interacting with software provides immediate feedback and thus leads to exploration, experimentation, and self-evaluation. Furthermore, designing programs produces useful and fun things, which vastly increases the sense of accomplishment when compared to drill exercises in mathematics. On the other hand, program design teaches the same analytical reading and writing skills as English. Even the smallest programming tasks are formulated as word problems. Without critical reading skills, a student cannot design programs that match the specification. Conversely, good program design methods force a student to articulate thoughts about programs in proper English.

But a lot on uncertainty exists as to the proper way to get educated. The reasons are plenty, but an educated guess would probably involve the following reasons:

1. Too much disconnected information, sometimes conflicting with each other.

2. An emphasis on the final product and not the process in most of the "official" materials. Harvey [1991]

3. A lack of explicit thinking tools for dealing with semantic complexity, and not syntactic detail. Felleisen et al. [2002]

The contributions I try to accomplish here are:

1. A central and coherent roadmap one can use to become a competent programmer by developing a taste for good programs

2. A review of the literature involved in the roadmap, pointing out their perceived strengths and weakness

3. Some indications on where to go after the way is walked

2 The Problem

The greatest problem that I try to partially address in this work is mentioned by Kurzweil [2006]:

> Most education in the world today, including in the wealthier communities, is not much changed from the model offered by the monastic schools of fourteenth-century Europe. Schools remain highly centralized institutions built upon the scarce resources of buildings and teachers. The quality of education also varies enormously, depending on the wealth of the local community (the American tradition of funding education from property taxes clearly exacerbates this inequality), thus contributing to the have/have not divide.

Not only that, but the quality of the teaching experience, by yielding to short term pressures of some market segments, are *dumbing down* their courses so they can "adjust to the times". Spolsky [2005]

With the advent of the Internet, the potential for emergent learning arises, both individually and communally, but in the sea of information exists a lot of pitfalls, possibilities for lost time and motivation and confusion.

3 The road map

Where do I plan to guide the reader? The shangri-la of this map is Papert's Principle:

> Some of the most crucial steps in mental growth are based not simply on acquiring new skills, but on acquiring new administrative ways to use what one already knows. Minsky [1988]

More specifically, I'll point to sources that present techniques to reach the same goals of Chakravarty and Keller [2004]

1. Convey the elementary techniques of programming (the practical aspect).
2. Introduce the essential concepts of computing (the theoretical aspect).
3. Foster the development of analytic thinking and problem solving skills (the methodological aspect).

3.1 Styles and Stances

What language to begin the journey? This is at the same time an important and a meaningless question.

It is important because all languages are designed with certain goals. My recommendation could not be stated better than Norvig [2001] does:

> - Use your friends. When asked "what operating system should I use, Windows, Unix, or Mac?", my answer is usually: "use whatever your friends use." The advantage you get from learning from your friends will offset any intrinsic difference between OS, or between programming languages. Also consider your future friends: the community of programmers that you will be a part of if you continue. Does your chosen language have a large growing community or a small dying one? Are there books, web sites, and online forums to get answers from? Do you like the people in those forums?
> - Keep it simple. Programming languages such as C++ and Java are designed for professional development by large teams of experienced programmers who are concerned about the run-time efficiency of their code. As a result, these languages have complicated parts designed for these circumstances. You're concerned with learning to program. You don't need that complication. You want a language that was designed to be easy to learn and remember by a single new programmer.
> - Play. Which way would you rather learn to play the piano: the normal, interactive way, in which you hear each note as soon as you hit a key, or "batch" mode, in which you only hear the notes after you finish a whole song? Clearly, interactive mode makes learning easier for the piano, and also for programming. Insist on a language with an interactive mode and use it.
>
> Given these criteria, my recommendations for a first programming language would be Python or Scheme. But your circumstances may vary, and there are other good choices. If your age is a single-digit, you might prefer Alice or Squeak (older learners might also enjoy these). The important thing is that you choose and get started.

But why it is a meaningless choice? Because the true skill of a programmer

is not to understand one language or another. As Papert [1981] puts it (he uses children, but they are just special example case):

> By deliberately learning to imitate mechanical thinking, the learner becomes able to articulate what mechanical thinking is and what it is not. The exercise can lead to greater confidence about the ability to choose a cognitive style that suits the problem. Analysis of "mechanical thinking" and how it is different from other kinds and practice with problem analysis can result in a new degree of intellectual sophistication. By providing a very concrete, down-to-earth model of a particular style of thinking, work with the computer can make it easier to understand that there is such a thing as a "style of thinking." And giving children the opportunity to choose one style or another provides an opportunity to develop the skill necessary to choose between styles. Thus instead of inducing mechanical thinking, contact with computers could turn out to be the best conceivable antidote to it. And for me what is most important in this is that through these experiences these children would be serving their apprenticeships as epistemologists, that is to say learning to think articulately about thinking.

Let me be more clear. The most important skill a great programmer possess is the capacity to think at different levels, and to think about thinking. Spolsky [2005] also makes this point:

> But beyond the prima-facie importance of pointers and recursion, their real value is that building big systems requires the kind of mental flexibility you get from learning about them, and the mental aptitude you need to avoid being weeded out of the courses in which they are taught. Pointers and recursion require a certain ability to reason, to think in abstractions, and, most importantly, to view a problem at several levels of abstraction simultaneously. And thus, the ability to understand pointers and recursion is directly correlated with the ability to be a great programmer.

Most texts ignore this aspect and instead choose to focus on the details of one programming language and/or paradigm. So in this paper I'll present only those that I judge that not only are aware of this epistemological fact but make it explicit.

4 The texts

4.1 How to design programs

This is the first text I'll recommend. The reasons are 2. First, they get the epistemological approach, as shown by statements as

> Learning to design programs is like learning to play soccer. A player must learn to trap a ball, to dribble with a ball, to pass, and to shoot a ball. Once the player knows those basic skills, the next goals are to learn to play a position, to play certain strategies, to choose among feasible strategies, and, on occasion, to create variations of a strategy because none of the existing strategies fits.
>
> A programmer is also very much like an architect, a composer, or a writer. They are creative people who start with ideas in their heads and blank pieces of paper. They conceive of an idea, form a mental outline, and refine it on paper until their writings reflect their mental image as much as possible. As they bring their ideas to paper, they employ basic drawing, writing, and instrumental skills to express certain style elements of a building, to describe a person's character, or to formulate portions of a melody. They can practice their trade because they have honed their basic skills for a long time and can use them on an instinctive level. Felleisen et al. [2001]

But the main reason is that they place an emphasis on a design recipe.

> Design recipes are the equivalent of soccer ball handling techniques, writing techniques, techniques of arrangements, and drawing skills. A single design recipe represents a point of the program design space. We have studied this space and have identified many important categories. This book selects the most fundamental and the most practical recipes and presents them in increasing order of difficulty. Felleisen et al. [2001]

Not only the book provides all that, they also couple it with its program environment and editor, a tool know as DrScheme. It is a professional as well as an educational leverage, and its use along with the text is imperative for getting the full experience.Findler et al. [2001]

4.2 The structure and interpretation of computer programs

This book is chosen because the authors clearly understand Papert's principle (they were both his students) by several comments as:

> Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming- language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems. Sussman and Abelson [1996]

> Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology – the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to." Sussman and Abelson [1996]

> Marvin Minsky and Seymour Papert formed many of our attitudes about programming and its place in our intellectual lives. To them we owe the understanding that computation provides a means of expression for exploring ideas that would otherwise be too complex to deal with precisely. They emphasize that a student's ability to write and modify programs provides a powerful medium in which exploring becomes a natural activity. Sussman and Abelson [1996]

Even the critics of the approach this text takes recognize its value and depth:

> Abelson and Sussman have written an excellent textbook which may start a revolution in the way programming is taught. Instead of emphasizing a particular programming language, they emphasize standard engineering techniques as they apply to programming. Wadler [1987]

But not all is swell with this classic text. SICP does suffer from some major deficiencies. Those are 2 according to Felleisen et al. [2002]:

1. ... sicp doesn't state how to program and how to manage the design of a program. It leaves these things implicit and implies that students can discover a discipline of design and programming on their own
2. SICP's second major problem concerns its selection of examples and exercises. All of these use complex domain knowledge.

Given those problems, I think it is best, specially for those without access to an experienced tutor/colleague, to leave SICP for a later visit, armed with the design tools that HTDP provides. Not only it will stress and test these tools, but it will make the road more bearable and enjoyable.

4.3 Further readings or The path to objects

4.3.1 Concrete Abstractions

This work is aimed at the same audience as HTDP and SICP, but with a different emphasis. It still uses the scheme language, which allows the use of DrScheme, a fact that gives it the same benefits as HTDP in this aspect.

It is also a textbook that gets the epistemological importance of the teaching, as this small description by its authors shows:

1. The book features thorough integration of theory and practice, and presents theory as an essential component of practice, rather than in contrast to it. Thus, students are introduced to the analytic tools they need to write effective and efficient programs, in the context of practical and concrete applications.
2. Significant programming projects are included that actively involve students in applying concepts. Each chapter ends with an application section, in which the concepts from that chapter are applied to a significant, interesting problem that involves both program design and modifying existing code.
3. The authors present development of object-oriented programming, one concept at a time. Each of the component concepts that constitute object-oriented programming (OOP) is introduced independently; they are then incrementally blended together.
4. In keeping with modern curricular recommendations, this book presents multiple programming paradigms: functional programming, assembly-language programming, and object-oriented programming–enabling the student to transition easily from Scheme to other programming languages.
5. The final chapter provides a transition from Scheme to Java. Providing this transition within a single book allows Java to be explained by comparison with Scheme, including showing an example program in both languages. Java also supports exploration of event-driven graphical user interfaces and concurrency.

or consider this self description directly from Hailperin et al. [1999]

> In summary, this book is designed to introduce you to how computer scientists think and work. We assume that as a reader,

> you become actively involved in reading and that you like to play with things. We have provided a variety of activities that involve hands-on manipulation of concrete objects such as paper chains, numbered cards, and chocolate candy bars. The many programming exercises encourage you to experiment with the procedures and data structures we describe. And we have posed a number of problems that allow you to play with the abstract ideas we introduce.
>
> Our major emphasis is on how computer scientists think, as opposed to what they think about. Our applications and examples are chosen to illustrate various problem- solving strategies, to introduce some of the major themes in the discipline, and to give you a good feel for the subject.

The use of a different mix of exercises, a different philosophy and the explicit tackling of object orientation and a transition path from scheme to another language makes this an choice to check instead or along SICP. Again, the application of the design recipes is something that enriches the experience in my opinion.

4.3.2 Smalltalk, Objects and Design

I present this book as the best source to learn object orientation that I have ever found. Considering that objects play such a huge importance on the software industry in our times, I think a suggestion focused on that would be reasonable.

I think this book is very good because it also gets the epistemological stance of Papert [1981], as the following extracts from Liu [1996] show:

> The goal is to design more like veteran software do. They choose among alternatives quickly and subconsciously, drawing upon years of experience, something like chess grandmasters choosing among moves. Lacking this experience, novices have a hard time discovering plausible alternatives, and an impossible time discovering subtle ones. For their sake then, I often argue alternatives and the trade-offs between them, so that they will have an outside chance of considering the same design the veterans do.
>
> When you finish the book, I hope you will be able to think about software problems in some of the ways that the veterans do, and be able to implement your thoughts in Smalltalk. Not expertly, however. Mastery of both object-oriented design and Smalltalk

comes only with actual practice. Of course, these are trims for many activities, from driving a car to playing the piano. But Smalltalk, more than most software tools, requires you to plunge in and abandon yourself to the language and environment. A taste for adventure definitely helps, more that in learning how to drive a car.

5 Conclusion

I tried to convey an adequate way that if followed would benefit a person and prepare it to became a competent programmer, not only in the sense of being able to dish out code, but to think at several abstraction levels simultaneously.

What I wish to stress is that the *mastery* of this concepts takes time and practice Liu [1996], as much as a decade according to Norvig [2001]. But to maintain a steady progression one needs to develop a *design taste*. That is what I hope the path outlined here would provide.

To wrap the article, I wish to cite Graham [2002] words on taste:

> Mathematicians call good work "beautiful", and so, either now or in the past, have scientists, engineers, musicians, architects, designers, writers, and painters. Is it just a coincidence that they used the same word, or is there some overlap in what they meant? If there is an overlap, can we use one field's discoveries about beauty to help us in another?
>
> For those of us who design things, these are not just theoretical questions. If there is such a thing as beauty, we need to be able to recognize it. We need good taste to make good things. Instead of treating beauty as an airy abstraction, to be either blathered about or avoided depending on how one feels about airy abstractions, let's try considering it as a practical question: *how do you make good stuff?*

References

Manuel M. T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.*, 14(1):113–123, 2004. ISSN 0956-7968. URL http://dx.doi.org/10.1017/S0956796803004805.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. An Introduction to Computing and Programming*. The MIT Press, Cambridge, Massachusetts London, England, 2001. MIT – Massachusetts Institute of Technology – Online version 3^{rd} edition 22. September 2002: http://htdp.org/2003-09-26/Book/curriculum.html – last visited 1^{th} Oct 2009.

Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum, 2002.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 12:369–388, 2001.

Paul Graham. A taste for makers. February 2002. URL http://www.paulgraham.com/taste.html.

Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. PWS Publishing Co., Boston, MA, USA, 1999. ISBN 0534952119.

Brian Harvey. Symbolic programming vs. the a.p. curriculum. *The Computing Teacher*, 18, 1991.

Ray Kurzweil. *The Singularity Is Near: When Humans Transcend Biology*. Penguin (Non-Classics), 2006. ISBN 0143037889.

Chamond Liu. *Smalltalk, objects, and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-268335-0.

Marvin Minsky. *Society of Mind*. Touchstone Pr, touchstone. edition, 1988. ISBN 0671657135. URL http://www.amazon.com/Society-Mind-Marvin-Minsky/dp/0671657135/.

Peter Norvig. Teach yourself programming in ten years. 2001. URL http://norvig.com/21-days.html.

Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, January 1981. ISBN 0465046274.

Joel Spolsky. The perils of javaschools. 2005. URL http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html.

Gerald Sussman and Harold Abelson. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, July 1996. ISBN 0262011530. URL http://mitpress.mit.edu/sicp/full-text/book/book.html.

Gerald Jay Sussman. Why programming is a good medium for expressing poorly understood and sloppily formulated ideas. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 6–6, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. URL http://doi.acm.org/10.1145/1094855.1094860.

P. Wadler. A critique of abelson and sussman or why calculating is better than scheming. *SIGPLAN Not.*, 22(3):83–94, March 1987. ISSN 0362-1340. URL http://dx.doi.org/10.1145/24697.24706.