

Chapter X

Toward a Nation of Educoders: A Roadmap for Sustainably Broadening and Improving Open Source Educational Software

Douglas L. Holton

Introduction

The goal of this chapter is to review the history of educational software development and to propose strategies for expanding the development of future educational software that is free, open source, and more effective for learning. These strategies would on the one hand make it easier for teachers, students and other non-programmers to modify or design their own educational applications, and on the other hand better educate software programmers about how people learn and how to design more user-friendly interfaces.

History of Educational Computing

Fischer (2005) points out that in the middle ages people were dependent on scribes to write down their thoughts and stories and also to read material written by others, and today many are in the same situation with respect to digital media - unable to express themselves via the creation of interactive computational resources, and instead dependent on products created by “high tech scribes”, or programmers and engineers. This is similar to the modern day issue of the digital divide, the split between the technological haves and have-nots, except that now the concern for educators is how we can help students gain computational fluen-

cy, or “knowledge that will position them to design, create, invent, and use the technologies to enhance their personal lives and social worlds” (Barron, 2004, p. 1). Fischer goes on to argue that the monopoly of “high tech scribes” needs to be eliminated by de-professionalizing software development, just as the monopoly of scribes were eliminated during the Reformation of Europe. There still will be a place for professional programmers, just as there is still a place for professional writers, however we should explore how to making computing more accessible to the general public. “Programming,” as Marc Prensky puts it, “is the new literacy” (2008).

That is one goal of educational computing. Begun in the early 1960s, educational computing is an effort to facilitate and improve teaching and learning via the development of software applications and technologies. “Educoders” refers to people who code, or program, educational software. The history of educational computing is one which has followed two strands that echo the scribe model and the de-professionalized model of digital resource development: computer-aided instruction (CAI) and end-user development.

Computer-Aided Instruction (CAI)

CAI, which was pioneered by people including the philosopher Patrick Suppes and Donald Blitzer (inventor of the Plato learning system), involves professional programmers creating software-based learning experiences for students. In early CAI tools, sometimes called teaching machines or programmed instruction, the computer controlled the pace and sequence (typically linear) of learning. More sophisticated modern CAI tools give the student or the teacher more control. Modern systems include intelligent tutoring software, hyperlinked training materials, and simulations.

Both early and modern CAI systems have shown strong learning gains in students and improved academic productivity (Molenda, 2008, 2009). Students benefit from the immediate feedback from the computer and the ability to learn on one's own.

There have been criticisms and limitations of the CAI approach which led in part to the formation of the end-user development strand of educational computing, as embodied by such tools as Logo and its derivatives such as Netlogo, BASIC and its derivatives such as Visual Basic, HyperCard and its derivatives such as SuperCard and AppleScript, spreadsheets, and computer-aided design (CAD) systems (Papert, 1993; Nardi, 1993). Problems with CAI may be grouped in at least three ways: cultural, humanistic, and pragmatic. Culturally, individualized computer-based instruction does not mesh well with school culture in which students learn together, at the same time, and under a teacher's direction. Teachers and college instructors also are resistant to be "handed down" new educational software to use without their input or customization. Ethically or humanistically, some CAI tools may treat students as objects to be trained (as in behaviorist animal training) and decompose learning into bite-size pieces (Papert, 1993, p. 164). Some topics, especially higher order concepts, cannot be so easily decomposed.

The main focus of this chapter however will be on pragmatic limitations and challenges to CAI, which are again related to the "high tech scribes" problem. There are not enough professional developers to meet the needs of each student and each teacher across the variety of contexts for learning in preschool, K-12 schools, higher education, and all the informal contexts for learning such as museums, at-home learning, and game-based learning. CAI is also susceptible to market forces, which essentially crashed the educational software market in 2001 (Reitel, 2005). This is

not to argue that CAI is a failure or should be abandoned, but that CAI might be combined with end-user development to broaden its reach, increase its effectiveness, and address some of the other cultural and humanistic concerns.

End-User Development

End-user development puts software development in the hands of non-professionals, such as teachers and students. Two pioneering efforts in this vein were the development of BASIC in 1964, and Logo in 1967.

BASIC

In the early 1960s, computers were programmed in highly specialized programming languages that were designed to optimize the efficiency of the program in terms of speed and memory use. With the development of time-shared mainframes and more affordable personal computers, however, efficiency was no longer such a hard constraint. BASIC was developed in 1964 by John Gemeny and Thomas Kurtz to allow individual students to be able to create their own programs for time-shared computers. The eight design principles for BASIC were (from “BASIC,” 2009):

1. Be easy for beginners to use.
2. Be a general-purpose programming language.
3. Allow advanced features to be added for experts (while keeping the language simple for beginners).
4. Be interactive.
5. Provide clear and friendly error messages.
6. Respond quickly for small programs.
7. Not to require an understanding of computer hardware.

8. Shield the user from the operating system.

What is of more note also is that the designers of BASIC released it to the public for free. “They also made it available to high schools in the Dartmouth area and put a considerable amount of effort into promoting the language. As a result, knowledge of BASIC became relatively widespread” (“BASIC”, 2009). BASIC had a built-in code editor and interpreter unlike other languages at the time. BASIC is generally case-insensitive and uses common English keywords instead of acronyms or symbols.

By any measure BASIC was a success and has even helped propel the computer revolution. Bill Gates and Paul Allen, founders of Microsoft, got their start creating and selling versions of BASIC. BASIC became widespread in schools during the 1980s as it was bundled with early Apple and IBM PCs. Many derivatives and modern off-shoots of BASIC emerged, including Visual Basic, which was the most popular programming language throughout the 1990s and is still fourth most popular (TIOBE, 2009).

Logo

Seymour Papert in 1967 helped develop the Logo programming language as an alternative to both BASIC and CAI (Papert, 1993, p. 163). Logo employed what Papert referred to as a “body syntonic” approach to programming, embodied either by a “turtle” on the screen or a robot which students could control by issuing Logo commands. “Turtles are said to be body syntonic, in that understanding a turtle is related to and compatible with learners' understandings of their own bodies” (Travers, 1996). That is, the turtles and robots could move forward, backward, turn, and do other things we could do physically.

Logo was designed to help children and beginners learn how to think computationally and learn about

computing for its own sake. It was not designed as a general purpose application development language as BASIC was. Hence, Logo and its derivatives such as Starlogo, Netlogo, Lego Mindstorms, and Scratch have been successful at helping students learn about computing and create animations and certain niche applications. Scratch is the most recent derivative which has been particularly successful at getting young kids engaged in creating games and animations. Netlogo has also emerged as a popular niche development tool, particularly for researchers in science education.

HyperCard and Visual Basic

Even though BASIC was designed to be a general purpose application development language, the space of computing expanded in leaps and bounds such that by the 1980s it no longer could be used to develop more modern, graphical user interface (GUI) applications. In the late 1980s and early 1990s, Visual Basic and HyperCard emerged as alternative, beginner-friendly environments that one could use to develop GUI applications for the Windows and Macintosh desktop platforms, yet the space of computing kept quickly expanding to encompass new platforms such as the World Wide Web and cell phones, rendering Visual Basic and HyperCard more and more obsolete. Visual Basic was also an expensive commercial product of Microsoft targeted primarily for business employees and software professionals, and Apple decided to no longer bundle HyperCard with its Macintosh operating system. Newer derivatives of both tools such as Visual Basic .NET and SuperCard still suffer from some of the same issues of costliness and lack of platform support, and some derivatives such as Visual Basic .NET also add a new level of complexity, making them too difficult for beginners to casually pick up and learn. Modern programming languages are designed well for scaling up to handle all these platforms and the needs of

professional programmers, yet they are not designed to scale down to the needs of beginners and non-professionals.

Current State of Educational Software

The Educational Software Market and Funding

As a result of being stuck in a “high tech scribes” culture of educational software development dominated by the CAI approach, educational computing has not had as strong an impact on education as many would have hoped, and the educational software market has been more susceptible to commercial market forces, either directly or indirectly. Directly impacting educational software in 2001, the commercial educational software market crashed from \$500 million dollars a year to less than \$152 million (Richtel, 2005), relegating it to a smaller, niche market. Indirectly, the development of educational software is impacted by economic forces and changing priorities, such as for example reductions in funding for educational software and curriculum development projects. In 1998, out of the \$30 billion dollar education budget, only \$30 million, or 0.01 percent was spent on research of any kind (Burkhardt & Schoenfeld, 2003). And Tinker (2006) points out that only one ninth of one percent of the National Science Foundation's budget in 2006 was spent on the development of new learning materials.

Projects which do receive funding to develop educational software and curricula suffer from sustainability and scalability issues. As one example the Carnegie Open Learning Initiative developed a highly effective online statistics course with supporting software that resulted in students learning more in half the time. But this effort has taken a number of years and millions of dollars in funding just in support of this one course. Other educational development projects at

other institutions have typically been abandoned as soon as the funding ran out. This is not a sustainable and scalable model for reforming and improving education as a whole because it still suffers from the “high tech scribes” problem.

Open Source Educational Software

The open source software movement is considered by many to be a key to moving education forward. Open source involves licensing one's software in a way that others may use it and redistribute it and even modify it, often with no cost involved. This movement has also spawned a parallel “open education” movement involving the open licensing of non-software curriculum materials such as textbooks and videos of instructor lectures. Educational software, however, still has an important key role to play in improving courses that cannot be achieved simply by opening up textbooks and lectures. The National Center for Academic Transformation (NCAT) for example has the goal of re-designing college courses to improve their learning effectiveness. They state that “successful course redesign that improves student learning while reducing instructional costs is heavily dependent upon high-quality, interactive learning materials” (i.e., software, NCAT, 2009). The limitation of NCAT's efforts is that is not involved in the actual development of such interactive learning environments, which is why their workshops are typically centered on instructors of common core classes for which some educational software already exists.

Although some end-user development tools are free and open source (such as Scratch), the open source educational software space has leaned much more heavily to the CAI side, with most software being created by professional programmers and computer scientists. There are two side-effects of this situation for open source educational software. One is the “STEM

effect”: the closer an educational topic area is to computer science, the more likely one will find open source educational software targeted to that area. Faculty and instructors and students in science, technology, engineering and math (STEM) areas are more likely to be able to learn and develop software using professional programming languages such as Java, C, and PHP (the top three programming languages according to TIOBE, 2009). The second side-effect is the “Hangman effect”: programmers, with little or no training about how people learn or concern with the curriculum standards of formal education, tend to develop more traditional, behaviorist educational software that resembles the kinds of activities they did when they were students, including such things as hangman games, typing tutors, drill and practice software, and electronic flash cards. Looking at the list of educational software offerings by open source education projects such as Edubuntu, Sugar, and OpenSuse Education/Live confirms this, but again this is not to say that these efforts are failures or should be abandoned, only that the space of education is far wider and deeper than a small band of open source “high tech scribes” for education can tackle.

Roadmap for the Future

Acknowledging the history and constraints faced in tackling this problem of improving education with the aid of educational software, the remainder of this chapter makes three specific proposals for moving forward in a broad and sustainable way. These proposals include:

- creating a short, interactive open course on how people learn and the difficulty of teaching for understanding and designing usable software interfaces

- creating a beginner friendly application development environment that works on more of today's computing platforms
- creating or building onto a community website for sharing and rating and categorizing open source educational software

There are also some proposals not touched on here which would no doubt help but are beyond the control of small groups of individuals, and require systematic action by governments and educational institutions. This includes the incorporation of more technological and computational literacy activities in K-12 education as proposed by the National Academy of Engineering and National Research Council (Katehi, Pearson, & Feder, 2009).

An Open Crash Course on How People Learn

What a teacher or educational software product presents to a student and what the student learns or even perceives can be two very different things. This discrepancy even catches many seasoned teachers by surprise, let alone software developers or college instructors untrained in learning theory. There are many ways to make instructors and designers more aware of the difficulties of getting a student's attention and interest and facilitate their learning and understanding. Two examples the author has used include demonstrations of *change blindness* and the video "Minds of Our Own", neither of which are openly licensed, however. Change blindness demos present an animation or picture in which something dramatically changes right in our field of view, and yet we do not perceive it until someone points it out to us or after an extended period of viewing. The "Minds of Our Own" video and an earlier video known as "A Private Universe" give extensive examples of everyday knowledge that not only do we often not know, but even many college engi-

neering graduates from top colleges surprisingly do not know. As the opening example in the video, students from two top engineering and science universities were handed a bulb, battery, and a wire on the day of their graduation and many were unable to make the bulb light. This was not a trick question, and one could no longer use excuses such as the students were “dumb” or the school was “weak.” These videos force one to acknowledge the difficulties of teaching for understanding and to rethink how we are teaching some of these topics if the result is “textbook” knowledge that does not transfer to everyday situations or other real-world situations.

Creating an openly available short course on this topic with newly designed openly licensed materials and videos and interactive software demonstrations may help software designers, instructors, and developers of any sort design better educational software. It has been shown for example in the domain of physics that “faculty involved in, or informed by physics education research, consistently post higher student learning gains than less-informed faculty” (Pollock & Finkelstein, 2008). Merely reading about educational research may help one become a better teacher, instructional designer, and educational software developer. A course designed with this purpose in mind could be even more effective; however it would require creating new animations and demonstrations and videos, some of which might mimic the non-openly licensed alternatives such as described above.

Beginner-Friendly Application Development Environment

Following the same eight design principles as BASIC, it is not infeasible to create a beginner-friendly programming language that can be used to create real-world educational applications that run on many of today's modern computing platforms, including web applications and applets, and desktop and cell phone applica-

tions. The Java software development platform from Sun in particular seems well suited to serve as a basis for creating such an environment, as the Java virtual machine (JVM) can run cross-platform desktop applications, web applications, web applets, and cell phone applications, including modern smartphones that run on Google's Android platform (which runs on Dalvik, which is nearly identical to the JVM). The problem with the Java platform for furthering educational computing lies with the Java language itself, as it was designed with software development professionals in mind only, and has a steep learning curve that is too much of a hurdle for most prospective casual developers including teachers and students (aside from the aforementioned "STEM effect").

Sun began to address this issue in 2006 with Project Semplice, an effort to port Visual Basic to the Java platform. However, this effort was abandoned. The project was later informally released with an open source license by Google under the name "simple" (2009), yet it only works with Google's Android platform. Another issue is that simple is not an evolution of Visual Basic but an exact copy of Visual Basic, with no changes to make it more beginner-friendly for example or to incorporate modern programming concepts such as closures.

HyperBasic is an open source language and development environment in the works, targeted to beginners and hobbyist developers. It builds on the open source Scala programming language and compiler which can compile applications for the Java and Android platforms. The language builds on the rich features offered by the Scala language (such as traits, actors, and closures) with simplifying features meant to aid beginners such as case insensitivity, a large top-level library of common procedures (as in BASIC and Visual Basic), and a library of sample games and educational applications. Unlike other compiler projects the goal is not

theoretical or to aid teaching computer science concepts for their own sake, but has the pragmatic goal of making it easier to develop software for today's computing platforms, the same way HyperCard and Visual Basic did in the 1990s. HyperBasic is due for release in summer, 2009.

Community Portal for Sharing and Finding Open Source Educational Software

The open source community has been served by website portals for sharing their software and finding and rating other software, including SourceForge, Google Code, and Github among others. The educational software community however has not been as lucky. There is no “education” section of SourceForge or the other sites, and thus Eduforge was created specifically to host open source educational software. However, this site is not well known to many developers, and most educational developers continue to use the other sites or their own to host their software projects. Eduforge is currently being redesigned and may better serve the education community in the future.

Another strategy is to bundle the hosting portal with the development environment. Both Scratch and Netbeans are examples of this. Scratch has a web portal where kids can share and rate the animations, multimedia, and games they create and has been very successful. The newest version of the Netbeans Java integrated development environment (IDE) features integration with the Kenai software hosting site.

While one must always be careful when creating open source community sites of the “built it and they will come” fallacy, HyperBasic will be designed with a complementary web portal similar to the Scratch website. This portal is being built atop the Drupal content management system, which has served as the platform for other web-based software created by the author in-

cluding Department 2.0, which provides a web presence for academic departments and schools but also serves as a platform for developing student professional identities via portfolios, blogs, and group wikis.

Conclusion

Alan Kay once said that the best way to predict the future is to invent it. Hopefully this chapter serves to illustrate that the future of open source educational software and its role in transforming education is feasible and “inventable.” Larger challenges still remain, such as how to teach computational literacy to more students at younger ages, yet there are other paths that small groups of individuals with little or no funding can achieve. This chapter proposed three strategies in particular: creating an open course on how people learn, creating a modern yet beginner-friendly development environment, and creating a web-based community for sharing and finding open source educational software.

References

- Barron, B. (2004). Learning ecologies for technological fluency: Gender and experience differences. *Journal Educational Computing Research*, 31(1), 1-36.
- BASIC. (2009). In Wikipedia, the free encyclopedia. Retrieved June 15th, 2009, from http://en.wikipedia.org/wiki/BASIC_programming_language
- Burkhardt, H., & Schoenfeld, A. H. (2003). Improving educational research: Toward a more useful, more influential, and better-funded enterprise. *Educational Researcher*, 32(9), 3-14.

- Dot.com bubble (2009). Retrieved June 15th, 2009 from http://en.wikipedia.org/wiki/Dot-com_bubble
- Fischer, G. (2005). Computational literacy and fluency: Being endependent of high-tech scribes. In J. Engel, R. Vogel, & S. Wessolowski (Eds.), *Strukturieren - Modellieren - Kommunizieren. Leitbild mathematischer und informatischer Aktivitäten*, Franzbecker, Hildesheim (pp. 217-230). Retrieved June 15th, 2009 from <http://l3d.cs.colorado.edu/~gerhard/papers/hightechscribes-05.pdf>
- Katehi, L., Pearson, G., & Feder, M., Eds., (2009). *Engineering in K-12 Education: Understanding the Status and Improving the Prospects*. National Academies Press. Retrieved September 10th, 2009 from http://books.nap.edu/openbook.php?record_id=12635
- Molenda, M. (2008). The programmed instruction era: When effectiveness mattered. *TechTrends*, 52(2), 52-58.
- Molenda, M. (2009). Instructional technology must contribute to productivity. *Journal of Computing in Higher Education*, 21(1), 80-94.
- Nardi, B. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: The MIT Press.
- NCAT (National Center for Academic Transformation), (2009). *Colleagues Committed to Redesign*. Retrieved June 15th, 2009 from <http://www.center-rpi.edu/RedesignAlliance/C2R/C2RAppGuide.htm>
- Papert, S. (1993). *The Children's Machine: Rethinking School in the Age of the Computer*. New York: BasicBooks.

- Pollock, S.J., & Finkelstein, N. (2008). Sustaining educational reforms in introductory physics. *Physical Review Special Topics - Physics Education Research*, 4(1).
- Prensky, M. (2008). Programming is the new literacy. *Edutopia*. Retrieved June 15th, 2009 from <http://www.edutopia.org/programming-the-new-literacy>
- Richtel, M. (2005, Aug. 22nd). Once a Booming Market, Educational Software for the PC Takes a Nose Dive. *New York Times*. Retrieved June 15th, 2009 from <http://www.nytimes.com/2005/08/22/technology/22soft.html>
- simple (2009). Retrieved September 10th, 2009 from <http://code.google.com/p/simple/>
- Tinker, R. (2006). NSF and K12 Reform. Retrieved June 15th, 2009 from <http://blog.concord.org/archives/14-NSF-and-K12-Reform.html>
- TIOBE Programming Community Index. (2009). Retrieved June 15th, 2009 from <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
- Travers, M.D. (1996). Programming with agents: New metaphors for thinking about computation. Unpublished Ph.D. Thesis. MIT, Cambridge, MA. Retrieved June 15th, 2009 from <http://alumni.media.mit.edu/~mt/diss/index.html>

Author Information

(to be included in final version of chapter)