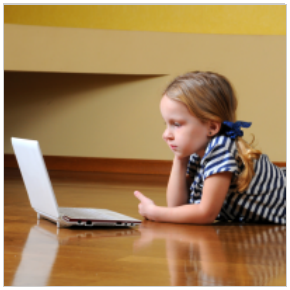
Electronic Engineering JOURNAL

Feature Article



February 4, 2014

Teaching Programming

The Case For

by Kevin Morris

There is some debate these days about teaching our kids to "code" and integrating software development into the mandatory curriculum along with reading, writing, and arithmetic. Many in our industry feel that this trend is misguided - including our own Dick Selwood, who **recently wrote a piece making the case that programming is the wrong thing to be teaching (http://www.eejournal.com/archives/articles/20140130-coding/)**. The arguments range from pointing to some of the very visible recent failures of software-based systems to worrying that programming is a passing fad that could leave our youth prepared for an obsolete profession.

I strongly disagree.

Software-based systems fail because modern software is the most complex creation ever attempted by humans. Programming is a multi-disciplined, massively-collaborative art that stresses every known attempt at complexity management, cooperation, verification, and human understanding. Software fails because nobody has yet figured out a way to make it never fail. And, the problem just keeps getting worse because, every single day, software becomes more complex and more difficult to create and verify.

The use of the term "coding" belies the flaw in the argument against teaching programming. "Coding" is a term derived from the "waterfall" method of developing software systems. That method prescribes development of a requirements document, production of a functional specification, and then "coding" that functional specification into machine-executable software using some programming language. "Coding" implies a process of simple translation - converting human-readable specification into something less accessible by humans but executable by machine.

However, the waterfall method of developing software systems ran out of gas some time ago. The complexity of the systems we are now creating in software is far too great for the specify-develop-test sequence to have any hope of reaching closure. The vastness of the requirements on a modern system would require volumes of specification to address. Our ability to detect flaws, holes, and inconsistencies in such a specification is near zero. Waterfall development is hopelessly reliant on humans' ability to wrap our brains around the entirety of a problem before proceeding to the next step. We are simply not that smart.

Now, the most complex systems tend to follow something resembling an agile or spiral development model. We assemble software systems by prototyping, evaluating, improving, and aggregating capabilities. We iterate - building successive prototypes until we reach a point of closure where a prototype meets a sufficient fraction of our needs. In these models, "code" is the language of requirements, specification, and implementation, as well as the dialect of choice for the evolution of that specification. Before one can begin to contribute meaning to that dance, fluency in the concept of programming (or "coding" - although I disagree with the use and implications of that word) is essential.

One does not write "King Lear" without first having a solid understanding of how to craft a complete thought into a written sentence, how to conjugate a verb, how to add meaning with expressive modifiers, and how to create rhythm that engages the reader. The basic lessons of spelling and grammar are essential, as are the strategies of storytelling. Before we begin expecting our youth to emulate Shakespeare, we must endow them with the basic skills of written communication.

The same is true for programming.

It doesn't really matter what programming language we teach, or whether the particular level of abstraction our children learn in grade school will be obsolete by the time they reach higher education. What matters is learning - at an early age - the process of beginning with a single snippet of understandable "code" and building from there into a system whose behavior is more complex than the creator can possibly contain in his mind. What matters is taking a real-world human problem and going through the mental process of turning that into a clear set of instructions for a soulless machine.

Practically everything I learned in college about EE is long obsolete. There is very little call for hand-assembling code for a Motorola 6809 8-bit processor. My mastery of Hollerith constants in FORTRAN 66 never gets put to the test. The uncanny ability I developed to locate bugs in TTL logic by probing a wire-wrap perf board is no longer the least bit impressive. However, the problem-solving skills I garnered by mastering that ancient craft have served me for an entire career. Regardless of what we study, the most important thing we learn is how to learn more.

Our world is increasingly reliant on software. Today, most people in the western world walk around with supercomputers in their pockets, interacting with and depending upon software on a near-constant basis. Software touches just about every aspect of a modern person's life, and this has never been the situation before in human history. In order to thrive in our world, people need to understand the current best thinking on how software works, when and why it fails, and how to make it better. If they do not comprehend the simplest principles of programming, they have no hope of flourishing in that future.

It does not matter whether these students go on to become programmers. Today, we accept the idea that children should learn reading, writing, and mathematics; arts and social skills - regardless of whether they will eventually become bankers, businessmen, artists, or engineers. These are the skills of surviving in modern society, not training for a trade. In a future world built largely of software, learning programming should most definitely be added to that list.

In the current explosion of the app-based economy, we see software applications for just about everything. The slogan "there's an app for that" aptly describes an environment where there literally are software apps being created to assist with just about anything one could want to do. The challenge is that programming is a required skill for creating such apps - along with expertise in the application area. I spent years in electronic design automation (EDA) - where one must have extensive knowledge of hardware engineering *and* software development. People who write apps for photo and video editing must be experts in photography or videography *and* software development. In fact, people developing applications for just about any field need software development expertise in addition to their core competence.

Not that long ago, "typist" was a valid profession. One could study the craft of typing, become fast and accurate, and make a career typing the work of others. Content creators did their creation and editing with pencil and paper or with dictation machines - then they handed their work off to a professional typist to

complete the job. Today, the vast majority of professionals type their own work. Keyboard competency has become a basic skill. In fact, with word processing - our content is normally created, evolved, and edited in typed form. It would be odd indeed today to find a professional who worked out documentation in longhand and then handed it off to a typist.

This same trend may well be happening with software development. The challenge of passing requirements or specifications from a subject matter expert to a "programmer" may be greater than the difficulty of teaching programming to the subject matter expert - particularly if that expert grew up in an education system where software development basics were part of the standard curriculum.

It has long been established that humans have a unique ability to acquire language skills at an early age. That ability declines significantly as we grow into adults. I will venture that programming has similar properties, and that children can acquire an innate fluency with communicating through code that far surpasses what they can learn if they first encounter the concepts as adults. We should teach programming early and universally.

We do need to teach more than just programming, though. The days of the lone programmer sitting in his or her basement and whipping up a world-changing application are rapidly coming to a close. Just about every useful system we will create will be the result of massive collaboration. Our children need to learn, at an early age, the social and management skills required to succeed in a highly-collaborative environment. The ability to work productively as part of a large team will be as important as understanding the programming language of the day. Probably more so.

A society in which children are not taught the basics of programming is one in which most of the citizenry depends on technology they simply think of as "magic." That lack of contextual comprehension lays the foundation for misleading mythologies, pseudo-religious manipulation of the masses, and general helplessness. It is extremely dangerous to have a situation where the most important concepts in the world are understood only by an intellectual elite. Programming should be ubiquitous, democratized, and embraced by the masses. That means our educational curriculum needs to reflect the essential nature of programming skills at a very fundamental level.