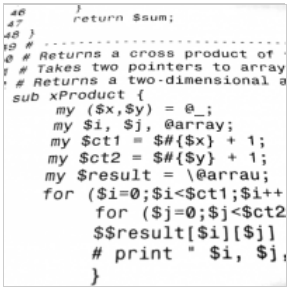
Electronic Engineering JOURNAL

Feature Article



January 30, 2014

Teaching Coding

Fast Track to Success or Failure to Get the Point

by Dick Selwood

Many years ago, I was attending a series of classes on computing. It was intended to introduce librarians, of which I was one, to the brave new world of "library automation." The first lecture was a description of the 80-column punch card and how holes in it represented characters. (For those of you lucky enough not to remember the era, we used to write code on coding sheets and then pass the sheets to a punch room where operators turned the sheets into card decks that were read by the computer. The program executed, and the results were printed out on continuous sheets of paper. You collected the paper, and with it, your coding sheet, and your card deck, you spent the next day or so trying to work out what had gone wrong.)

Knowing how the 80-column card stored information was interesting but gave no perspective on how computers could manage the library issue system or book catalogue.

This same misunderstanding of what computers are, their uses, and their implications for society and personal life is behind the rush to teach school children coding. This year, the British government has mandated that all pupils entering schools will, from the age of five onwards, be taught to write code. A few weeks ago, President Obama urged all young people to learn to code. Around the world there are similar initiatives. They have been greeted with joy by two groups of people: those who know nothing about computers, and programmers, who see this as a confirmation of their special abilities.

The reasoning behind these initiatives appears to be that computers are so important to everyday life that the entire population has to be able to program them. None of the discussion surrounding this debate seems to have looked at the fact that consumer electronics and PCs are the tip of the iceberg for processor deployment. None of the proponents seem to understand that writing code should be implementing a solution to a problem that has been defined and analysed and is not the solution itself.

In 2013 there were, around the world, some spectacular high-profile issues surrounding electronics and computer-based systems. In the US, perhaps the most public problem was the HealthCare.gov website. Its teething problems were not due to poor coding. They appear to have been due to a standard problem with big systems: politics. In this case, it was national politics that drove the system to go live without adequate testing. In Britain, other government systems are facing problems through the classic project management issues - ill-defined initial objectives, mission creep as people ask for extra features to be included, and pressures to deploy the system before testing is complete.

Testing in this context includes not just debugging code, but also testing the user interface, as the developers will have made assumptions about how the user will behave, which will, in field use, prove to be inaccurate – normal people don't always think in the same way as the people currently developing systems. (If you want to learn more about the problem, look for Alan Cooper's book, *The Inmates are Running the Asylum.* Although it is now nearly twenty years old, it contains a lot of good sense.) If it is a website, then stress testing the site to see how much traffic it will bear is also necessary.

None of these failing projects would have been improved by a wider knowledge of coding. And many of these projects are failing because the focus has been on implementing rather than on solving.

Knowing how to code will not give any understanding of how GCHQ and NSA are using computing power to store and analyse your internet activities and your phone calls.

Another flaw with the argument to teach everyone to write code is that conventional programming will, possibly within the next ten years, become a quaint anachronism. Look at parallel field chip design. (If you are familiar with this area, forgive me for this Janet and John description.) Chip design started with engineers drawing circuits using conventional symbols for devices – transistors, resistors and capacitors. These were given to drafts-people who turned them into the polygons that represented these devices in a silicon layout. These stages were replaced by schematic entry, where conventional symbols were drawn on a workstation and translated by the computer into the polygons. Chip Design had transformed into EDA – Electronic Design Automation. Later, the schematic entry was replaced by hardware description languages (HDL), a higher-level abstraction. Verilog and VHDL described, not transistors, but the logic functions that the circuit was to implement. The engineers using HDLs were effectively writing programs.

HDL is now largely replaced by Electronic System Level (ESL) tools, which use even more abstract languages, such as SystemC, to model the system that the chip is to implement. In some cases modelling tools, like National Instruments' LabVIEW or Mathworks' MATLAB, are used to build a graphic representation of the functions of the system that are needed, and a chain of tools converts these into the polygons for manufacture. As these tools have grown more abstract, so there has developed a range of verification tools that check that what is implemented is what the specification required, and test tools that measure how well the circuits actually work. Also, given the massive size of current designs, there is now a base of Intellectual Property – blocks of circuit design that carry out specific tasks. These may be for a whole processor, with ARM the clear market leader, or for interfaces, such as a USB controller. This means that the EDA equivalent of coding is less and less relevant.

The real challenge is not code writing; it is analysing the problem and designing a solution, for which the code is merely an implementation.

As was previously mentioned, the majority of the "teach children to code" enthusiasts have not realised that a huge amount of code is written for embedded systems - not for familiar IT type applications. Here again, there is the rise of model-based approaches to developing applications. The large, complex, and mission-critical systems that the aerospace and defence industries develop were the driver behind developing first UML (Universal Modelling Language) and then SysML (System Modelling Language). The model-based systems are linked to code generators and produce very high quality code.

A positive story in 2013 was the confirmation of the Higgs Boson. The Boson was detected by smashing two streams of protons into each other and looking at the particles that were created. The energy in these proton streams is enormous, and they have to be very tightly controlled. The software for running the controllers

within the Large Hadron Collider, the massive underground facility on the border of France and Switzerland, was developed using NI's LabVIEW graphic development system, so no coding skills were deployed.

Coding is not going to help people make better use of mobile phones and the apps that run on them. It is not clear how being able to code is going to help people reign in the efforts of security organisations to monitor the communications of all and sundry. For the relatively small percentage of the population that will enter the technology industry, there may be a marginal benefit in knowing a programming language. The religious wars on what language to teach are going to be great fun to watch, but none of them will help anyone do anything very useful.