A talk to the undergraduates in CS 337 on the last class day

Jayadev Misra

5/2/03

Let me talk a little bit about myself, a topic that I am slightly embarrassed to talk about. I started working with computers at a relatively early age in the late 60s. I was introduced to computers by a friend who showed me one evening how to write some simple programs in Fortran, the prevalent language of the day. It took him a little less than 2 hours to explain to me what computers (and tapes drives, memory, printer) are, what programming is, how one prepares a deck of punched cards (punching holes manually is not to be recommended) etc.; you see why I believe so strongly in study groups of your peers.

I was fascinated by computers for two reasons: First, you get the answer to an experiment immediately, and second, you make no approximations. It is only later that I had to revise my opinion considerably. After some practice I became proficient at programming, eventually designing a language, a compiler and an operating system for my Bachelor's thesis. I had learnt no theory, all my algorithms were my own. It took considerable thinking and experimenting, but there was a rush when it all worked. Programming was a great intellectual challenge, with an impartial referee. You struggled with your programs and you had a right to feel victorious when you at last wrestled one to the ground.

I had two great surprises immediately after I joined graduate school. I learnt the notion of grammars, and saw how I could have shortened my compiler considerably had I known about it earlier. And, I learnt about program proving, using assertional methods, which could establish correctness without doing any testing at all. I had been using something like invariants in my reasoning about my programs, without any good theoretical basis or understanding. Here was something truly marvelous. Those were two stunning examples of theory in practice, which could have avoided a large amount of wrestling with programs.

What has any of this got to do with you? This is another era, machines are faster by several orders of magnitude, we have a huge number of tools, including powerful debuggers and program analyzers, and we program on personal computers at our convenience. Gifted high school students hack out large systems by working and debugging hard. Who needs theory, particularly when they are not immediately applicable?

Well, you are all young and you can be forgiven for asking these questions. But there are many in our profession, including managers in large companies, who continue to believe that software is a contact sport. Design and analysis prior to coding is for sissies. My wife was offered a job after she had finished a Masters degree in CS, but she was offered the same salary as one with a bachelors degree, the rational being higher education adds nothing more to one's programming skills. It is instructive to recall the words of John Maynard Keynes, a prominent economist: Practical men, who believe themselves to be quite exempt from any intellectual influences, are usually the slaves of some defunct economist.

Let me try to draw an analogy with automobile tinkering. In the late 60s and early 70s, before the manufacturers put in electronic fuel injection and other such gadgets, tinkering with automobiles was a favorite pastime for many young and old men. Yet, no significant advance in automobile design has come by tinkering with it. Lean-burn engines, antilock braking, better suspensions, have all come from people who went to engineering schools, and learnt physics, thermodynamics and mechanical engineering. Tinkering might ignite a passion, but that is not an end in itself.

So it is, I contend, with software systems. It used to be the case that brilliant people designed software by intuitive leaps, and the rest of us could only marvel and try to emulate those practices. We have come a long way. The very first program by the great John von Neumann was a mergesort, a typical freshman-level programming exercise today. We ask students to develop compilers which would have been worthy of a team effort in a large company 30 years ago. And, conversely, the web browsers of today could not have been implemented with the software knowledge we had 20 years back, even if all the hardware were available then.

Today, higher level languages, including their type systems, methodologies for component integration, databases and query languages, specialized theories that are applicable for specific domains, have made software design an activity for trained professionals, not inspired amateurs. I can not imagine developing an algorithm without worrying about its asymptotic running time or if the problem itself is NP-complete. I can not imagine starting to write a program without a precise understanding of its specification.

Your working life will really begin in another 15 years, or so. I can not predict which skills will be in demand then. But I can't imagine that you will be heavily involved in coding and debugging. You will probably be designing software, several orders larger than what you do in a project. A first-rate school has to prepare you for such a future, not the one you will immediately confront after your graduation.

What are the short-term skills you need: Familiarity with some popular programming languages, APIs, ability to use operating systems, downloading and installing packages, and deciphering their manuals and applying them in a specific context. And, then there are software engineering skills: how to do a code walk-through, collaborate, communicate with people of different skills and manage projects. Many of these skills are transitory, and often specific to one company. We will be abdicating our responsibilities if that is all we taught you. Teaching students to be familiar with a specific package is common in lower-ranked schools; their concerns are more short-term and the students they educate are destined for organizations where the skills required are installing and maintaining packages. We expect that you will be the developers of packages. How will you design a language without knowing about grammars, networking systems without knowing some graph theory or message transmissions from spacecrafts without knowing error-correcting codes?

The longer term skills involve learning to understand the requirements, program design and analysis including the questions of correctness and performance, knowing where to look for extant solutions, program integration and testing. Designs of appropriate experiments, to measure system performance and for testing, are of immense importance. Then there is domain specific knowledge. If you are going to do computer-based animation you must know the theory and practice of computer graphics. And, you must know where to look for answers.

The computers you will program eventually may not be recognizable as a computer to many of us today. There are people who are implementing tiny operating systems and databases in devices with, say, 4k of main memory. There are people who are studying power consumption in smart cards, for possible security measures. And, developing web access from a cell phone was not even imagined 10 years back. True electrical engineers could not have believed in the 90s that they will have to know predicate calculus; yet, that is a routine activity now if you are applying model-checking in chip design.

It is difficult to predict the future; so, let me predict the past: Long term skills pay off. I hope we are doing our best to prepare you for the unexpected and I hope you will be a credit to this institution.