

Why do I teach? It's what I do. It is as simple as that. People don't choose their vocations, vocations choose their people. It was never a conscious decision, just a gradual realization over a period of years. As an undergraduate, if somebody had suggested to me that I would be doing this, I would have laughed at them. As a graduate student, I discovered that there could be more to it than reading out notes and copying them onto a blackboard while dozens of students struggle with the dilemma "should I just try to copy it all down, or should I try to understand it and hope that I'll be able to work out the details?", and the idea became less laughable. Later, I was often asked to take over classes for absent professors, sometimes with almost no warning, and discovered that it could be an interesting, challenging, and even rewarding task after all. Thus, I fell into teaching, without ever really noticing.

Why do I continue to teach? When it is very common for my students to graduate straight into jobs with salaries higher than my own, and within a few years they are earning twice as much. When I am afraid of the prospect of retirement because I can't see how I could possibly afford it (I often joke that I should take up smoking so I won't have to worry about it), and when I find that as, dare I say it myself, a very accomplished programmer and software engineer, I could easily and literally triple my income by moving into the "real world", why do I carry on?

The answer is that I can't bear the thought of doing anything else. Our young people bring with them an endless supply of strange philosophies, hopes, fears, and dreams, interesting ideas, and just-as-interesting misconceptions. They are certainly society's most interesting division, if only one can understand them and relate to them on their own terms.

That is where the hard work, and all the rewards, lie. Society changes so quickly that it is very difficult to really understand someone less than half of one's own age, to know what is on their minds, and what it really means, to know *where they are coming from*. But I do believe that knowing students is an essential prerequisite to really teaching them, as opposed to simply filling them with facts.

If I can help to relight the fire, bring back the sense that there are things worth exploring, the sense of wonder that the cruel and unusual

punishment of high-school knocked out of them, then I have done something worth-while. As time goes on, I realize that I have probably missed the boat on marriage and children, but I have still got plenty of other people's children to share in. Changing profession would be like abandoning my family.

How can a person gain a deep understanding of their students? Homework, trickery, and honesty.

They homework is taking the time to follow youth culture. After all, we expect them to respect and adapt to ours, the least we can do is to be aware of theirs. I could almost say that I learned to speak "American" from the Simpsons, I have learned a fair amount about modern music, although I have not learned to like it, and I have even learned something of the technology of skateboarding. The list of opportunities is endless.

That homework is the preparation that makes the trickery possible. Some students are easy to speak with, some just won't stop, but increasingly many prefer to avoid all contact with faculty. I have made it my habit to ask each of the quiet ones at some point to come and speak to me in my office for some reason, perhaps about grades, perhaps about "something" that came up. When they arrive, we accidentally get side-tracked and talk about something else. It doesn't matter what, just something non-academic, to get a conversation underway. That is where the homework helps: if I can recognize a song hissing out of the headphones or a logo on a tee-shirt, that is usually enough. Just one thing in common, one thing that I know about, and they have a personal interest in, is almost invariably enough to get someone to open up. The aim is of course not to get them to reveal their secrets, but simply to "open a dialog", and become someone that they talk to, not just someone that they have to listen to. In the end, they often find it quite hilarious that I pretended to have some academic need for a talk: "Why would you imagine you need to trick me into having a chat?"

What of honesty? It is very tempting just so say *the right thing*: "It may not seem fair that the football team gets special treatment, but without them, your fees would be higher, and how would your scholarship be funded?", or "They have to restrict your internet access, in order to prevent piracy. Artists have to make a living". Making the effort to see things from their perspective, and being able to admit "yeah, that sucks", or even "I don't know", breaks down the last barriers, turning a professor from part of the establishment into a real person.

The purpose of all of this is to bring a bit of life into the classroom. Once the students have come to see me as someone they can talk to freely and openly without being judged or “set straight”, they will actually participate positively in classes, and that is not normally the Engineering Way. Explaining a new programming technique or a tricky system feature always requires some worked examples or case studies. Studying programs on the blackboard or overhead projector is not a very inspiring experience, but with a class that is willing to interact freely, it can be very different.

The idea is to set the scene, describing a real-world (-ish) problem that is in need of a software solution, and ask for suggestions for a plan. At this point, only a few eager students from the front rows will say anything, but that is to be expected, this is not the important stage. When the initial plans are made, and the students are all expecting to see the usual blackboard software development, everything stops. “OK, let’s make this happen”, bring down the projector screen, log in to the computer, and start live software development. Not just in front of the students, but with the students.

To a non-programmer, that will seem like a really obvious thing to do, and it will be surprising that it is generally never done. The reason is that programming is a very delicate operation. Even the best of us will inevitably keep on making tiny almost undetectable mistakes that can easily take hours to discover and put right. Every time I do this in class, I experience that little frisson of panic because it could quite possibly fall completely flat. This really does require total command of the material, as failure here could leave a whole class with the belief that it is just too hard to do.

The class proceeds simply by taking suggestions from the students, and putting what they say into the program. Of course, some care is required to ensure that they suggest things that will work, but that is not too difficult, and some exploration of wrong suggestions is also very helpful.

The aim is to have some basic functionality that can be tried out within about fifteen minutes, and to make sure that there is a mistake in there somewhere. Usually there are a few accidental mistakes anyway, but always one deliberate one. The first time I try to run the program and say “Oh, it didn’t work”, that is when the faces at the back of the classroom look up, to gloat over the disaster. But suddenly it’s real, this is just what happens when they write programs, and it’s the same for the

professor too. Soon everybody is joining in the fun, calling out the mistakes, and in no time, we've fixed it.

After that, just about everybody is taking part actively. They have seen that this is real programming, and that it is their own program, because only things suggested by the class get in there. By the end of the class, they have jointly produced some real working software, and seen how someone highly experienced with the system makes it work, and makes the same human mistakes as they do.

After the first time for each class, this method works much more smoothly. They know what is going to happen, and remember that they quite enjoyed it last time. For the few shadowy faces who always sit at the back, the second time might not be quite exciting enough to make them turn their game-boys off, but progress has been made, and a whole class knows *we can do it too*.

Apart from showmanship, there are a few other obvious things that I have very slowly learned over the years. One is that I have people at a very delicate stage. I am teaching them the most fundamental areas of their chosen major. For better or for worse, this is most probably what they are going to be doing for the next 45 years. To make it into something dull would be an unforgivable offence. Unfortunately, the teaching of programming has a strong tendency towards dullness. Many students come here with dreams of becoming video game developers or creating the next "killer application", but find themselves writing programs that ask them to type in ten numbers, after which they calculate and print out the average, or perhaps if they are lucky, a program that will calculate the area of a circle.

These things are certainly valid exercises in the techniques required for programming, but they do not engage the interest. The reason is that in the early days, all interactions with computers took the form of typed text, on punched cards, paper printouts, or teleprinters. Text books have not moved on from that model, and as a consequence neither have teachers. For twenty years now, virtually all computers have had nice sharp high-resolution multi-color video displays. Why should teaching have stayed so entrenched in the past? Because the interfaces to video displays, the things a program has to do in order to produce graphical output, are appallingly over-complicated. Even the simplest of tasks, just drawing a colored circle on the screen, requires at least a three-page program, every part of which is both incomprehensible and inexplicable to the beginning programmer.

I found that situation completely unacceptable. Even students who live and breathe computers are completely turned off by “the average is...” programs. So one summer some years ago, I pulled together all my resources, and created a totally new interface. One that makes producing graphical output from a program actually easier than text-based interactions. It was a very unpleasant job, Microsoft systems are notoriously poorly documented, and it consumed the entire summer. I nearly gave up on it quite a few times.

But as a result, I can let students do what they actually want to do. All of the essential exercises in programming techniques can be perfectly adapted to the graphical environment. Instead of just calculating parameters of geometric shapes, they make those shapes appear, and can see instantly whether their calculations were right or wrong. Instead of calculating the average temperature from a year of meteorological observations, they can plot the whole gamut of observations on a multi-colored graph. Instead of generating thousands of meaningless random numbers to practice large array handling, they can sample their own voices, distort them digitally, and play back the amusing noises that result. By the end of their first semester, students really do produce a working video game. I can walk into the lab, and find students playing with their home-works.

I suppose the proper lesson to draw from this is that I get out what I put in. It took a lot of unpleasant work to make it possible, but it is so much easier to teach a class when the students like what they are doing, and to have them enjoy their home-works was a benefit that I had never anticipated.

The lesson I really learned however, is that text-books really aren't very good, in the computing disciplines at least. Only *we* can really know our students, and only we can really produce the best material for them. Of course, a book for reference is a very important and useful thing, but a book to teach by is a very unsatisfactory arrangement. We are supposed to be experts in our fields after all.

Take full control of a class, every aspect of it. That would be my advice.