

High-Level Problems in Teaching Undergraduate Programming Languages

William Cook

University of Texas at Austin

I have been teaching programming languages (both undergraduate and graduate) for the last 5 years. My graduate class is going well, but the undergraduate class is a disaster. We have two versions of the undergraduate course: regular and honors. I have taught the honors course twice and the regular course three times. I have tried 3 different books (Mitchell, Scott, Tucker & Noonan), changed my teaching style completely (from PowerPoints to blackboard), and experimented in other ways. Some of these experiments have been very successful, while others have flopped. My teaching evaluations have gone down consistently since I started teaching the course. I am frustrated and unhappy. My students are too.

Before becoming a professor, I spent 10 years in commercial software development, first at Apple and then at several startups. My last startup grew to 160 people from 1998 to 2003. I was involved in hiring almost everybody, since I was one of the 4 founders. At one point I managed the 60-person engineering department, which included development, testing, documentation, and consulting. I was also the software architect for the company. One of the interesting things about this experience is that we very early on rejected traditional object-oriented programming and adopted a model-driven approach instead. As such, we were doing real-time programming language research, and surprisingly it worked out fairly well, because we had a team of very seasoned programmers. Our system used a wide range of languages, including C++, Java, JavaScript (in browsers on the server and in tools), HTML, SQL, XML, makefiles, in addition to custom languages for user interfaces, security, data modeling, constraints, workflow, and reporting. We felt that we used the best tool for each job, rather than a one-size-fits all approach of a general-purpose programming language.

Before that, I did a PhD on programming languages at Brown with Peter Wegner. I worked on the foundations of object-oriented languages. It was very exciting to help develop the theory for the OO paradigm, which had evolved successfully without much theoretical foundation for over 20 years. Objects are now 40 years old, but they are in a strange state of at the same time being relatively mature but also still widely misunderstood, both in the academic community and in the industry as a whole.

In thinking about this situation, I believe I have a good sense of some of the problems we face and a few potential opportunities. But I don't have a solution. Here are some observations on the current situation:

Desensitization. When a person uses a tool to achieve some goal, they quickly learn how to work around the short-comings of the tool in order to get their job done. As a result, the person becomes *desensitized* to problems in the tool. Humans are very adaptable and this process happens very quickly. During the learning process, we tend to assume that problems are caused by our own inexperience with new situations and tools. We do not know enough about the problem and situation to analyze the tool and find its faults. Once we do know enough about the tool and its use, we are desensitized to its shortcomings. Desensitization is a general problem that affects software quality assurance teams, which must continuously and

consciously fight against desensitization in order to do their jobs. Desensitization is a subtle and pervasive problem in the area of programming languages. It is difficult to get students to think critically about languages once they have internalized the language and become desensitized to all the things that make using the language difficult. It is like asking someone to critique “English” and improve the language. This is not something we do easily. It is possible to introspect about our tools; using more than one tool is a good technique, as it provides some basis for comparison. However, it is essential to really learn the other tool first. Requires a change in mind-set from “working around problems” to “working on the tools”

Indoctrination. Indoctrination is a well-known problem: our tools influence our thinking about problems: “When you have a hammer, everything looks like a nail.” With languages the indoctrination is often so deep that we cannot imagine any other way of thinking about a problem. Learning a second language often requires a cultural shift, so that one does not just speak natively, but thinks in the native language. For programming languages, this requires extensive use of a new language. There are many old stories of programmers who try to use a familiar style in a new language, .e.g. writing FORTRAN-style programs in Lisp. But this problem is still pervasive. One of my favorite case is the running example in the book *Programming Languages: Principles and Paradigms*, by Tucker and Noonan. The program is written in Java, but it is pretty clear that it was translated from a language like ML. The resulting program is a bad example of both styles of programming (OO and functional).

Object confusion. The Tucker and Noonan example is also a symptom of a deeper problem: object-oriented programming is not very well understood, even by the authors of undergraduate programming language texts. They almost always conflate abstract data types and objects. I think this is partly because of the assumption that abstract data types are the only kind of data abstraction, so objects must be a variant of abstract data types. Perhaps I am too sensitive to this issue, because I have worked on it, but I have done some empirical studies of the question. Whenever I am in a group of PL researchers, I always try to ask them “what is the relationship between objects and ADTs”. The fact that a huge argument usually erupts is a sign that we don’t have much common understanding of even this basic notion. I have also studied textbooks, and they invariably fudge the issue. In doing so, they fail to address some of the fundamental issues in design of data abstractions (the ideas underlying the extensibility, or expression, problem).

Qualitative effect of Scale. Perhaps the most fundamental problem I have found in the undergraduate curriculum is that many of the more interesting issues in software development do not arise until systems (program size, problem size, or team size) reach sufficient scale. For example, formal verification and software engineering don’t seem very useful when students can hack out a program over the weekend to satisfy the requirements of a course project. Writing a fast algorithm versus a slow algorithm doesn’t make much difference unless the problem is large. This issue doesn’t arise, for example, in teaching medicine, because even a small issue can be a question of life or death. In programming languages, issues of modularity and reuse don’t matter very much for small programs. The problem, as I see it, is being able to expose students to these larger issues within the structure of an undergraduate curriculum.

Breadth versus depth. We have recently started a project to reconsider the entire undergraduate curriculum at UT Austin. A committee developed an overall approach, and we have explored it in a full-day offsite of the entire faculty. One key idea is that CS is now too broad to be able to teach students all of it, and any attempt to do so will take so much time that little room is left in the curriculum for in-depth study of specialized topics. We are attempting to build a curriculum around the idea of a “small core” that can be taught in the first two years (roughly). The students can then select a two or more specializations. The same problem might apply within a programming language class itself. There are lots of basic topics (binding, scope, etc) that might be considered essential knowledge, but covering them leaves little room for discussion of advanced topics like continuations, generic programming, type theory, partial evaluation, process algebra, monads, distributed programming, metaprogramming, reflection, and aspects. I have not figured out how to resolve the problem of breadth versus depth.

I want to give one concrete example of something that I did that worked really well in my undergraduate PL class. I assigned a problem, to be implemented in Java, entailed solving the extensibility problem (aka the expression problem). The problem involved creating a library that was extended in two ways, and then the results combined. Students had to show how the library was derived for each of the three extensions. Since there is no known pattern for implementing full extensibility in Java, the assignment was fundamentally unsolvable. I told the students that I was giving them an unsolvable problem, and that they have to make and justify their best approximation of the full solution. The students really enjoyed working on the problem and didn't seem to mind that it was insolvable.

I also wish that I could communicate to students that this is an exciting time in programming languages. There is a great deal of experimentation going on with new (and old) approaches that don't fit into the object-oriented mold. I believe that in the next 10 years objects may be overthrown as the leading paradigm for programming (you can all insert your own favorite back-bench paradigm as the next dominant paradigm). I want to talk about how we can prepare students for this kind of deep change.

My sense is that PL may be the common undergraduate course with the greatest variation in its content. As far as I can tell, we don't know what to teach our students, and there is no consensus on how to organize the course. I look forward to discussing these, and other issues, so that I can improve my own undergraduate PL course and also guide the development of the new UT curriculum in areas that touch on programming languages.