

Rethinking Pedagogy for Teaching PL with More than PL Concepts in Mind

Lori Pollock

Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

Depending on individual department goals, undergraduate computer science educators teach theoretical concepts and skills with varying priorities on preparing students for continuing to graduate school or embarking on a career that leverages their education immediately after college. Programming languages play a key role in that education for both student profiles, even in curricula that include no core programming language concepts course. The “fattening” of our discipline into many subareas has prompted curricula reviews and rethinking of the undergraduate CS content, including programming language concepts (PL) and where they are taught in the curriculum. This paper proposes that we also rethink the pedagogy for teaching PL with diversity in mind.

1. Introduction

As the field of computer science broadens to include increasingly more subareas and a more interdisciplinary nature, curricula committees are faced with making difficult decisions about essential core requirements and curricula design. In most cases, the computer science curriculum needs to address the educational needs of both the students who will immediately enter the workforce (pre-industry) and students who will continue their education in graduate school (pre-graduate).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The curricular debates focus primarily on which essential concepts and technical skills should be taught and what paths through the program are most appropriate for different students. Concurrently, the concern over low enrollments, and particularly recruitment and retention of women and other underrepresented groups has prompted significant discussion about the content and pedagogy for the first course sequence in the major. However, little discussion focuses on pedagogy of upper level courses, particularly that initiated by non-technical learning goals in preparation for graduate school or transitioning to software engineering positions.

This paper proposes rethinking the pedagogy for teaching PL design, implementation and application in light of learning goals for graduate school preparation, recent recommendations for CS curricula changes based on an observational study of new professional software developers (2), and recommendations for retention. Historically, programming language design, implementation, and application have been key components of almost every computer science curriculum due to the central role that programming languages play in humans controlling the computing systems. PL is well suited for achieving many of the learning goals for all kinds of student. By leveraging PL education as a vehicle for attaining non-technical learning goals, the PL course can serve multiple purposes beyond teaching PL, reaching more of the grander goals of a computer science education, and helping to justify PL as an important component in the curriculum.

2. Beyond Learning PL Design, Implementation, and Application

This section examines the curricula considerations to take into account based on studies that look beyond content in the CS curriculum. Three main objectives are considered:

- the needs that recent graduates exhibit when they join their first software development team
- the experiences that help students gain admission to and succeed in graduate school in CS&E, and
- the practical suggestions for recruiting and retaining women and underrepresented minorities in CS&E computing majors.

2.1 Pre-industry Students

Academics and the software engineering industry often dispute whether the goals of undergraduate education should focus on the technical skills for current industry needs or instead teach the major concepts and general problem solving skills using the most appropriate teaching tools with little consideration for what is used in industry. Regardless of the view taken, it is worth examining whether other pedagogical changes beyond the technical skills taught could help ease the transition from the academic world of computer science into the software engineering industry.

Begel and Simon (2) recently reported on their study of 85 hours observing eight new software developers in their first six months at Microsoft. The new software developers demonstrated many programming strengths and persistence in generating different hypotheses when they faced unexpected behavior while learning new tools. However, they also observed that new graduates struggled to be effective in *communication, collaboration, technical, cognition and orientation*.

The major communication problems were in knowing how and when to ask questions of others (i.e., asking questions soon enough and at the appropriate level). Collaboration issues involved working in large teams, working in conjunction with multiple teams, and working with a large, pre-existing codebase. The technical difficulties centered around having to adjust to the use of tools (e.g., test environments, debuggers, and revision control systems) for large-scale development and knowing the importance of leveraging the automation and support of the tools instead of relying on error-

prone human effort. New software developers found the process of taking notes during impromptu teaching sessions and team meetings, and collecting and organizing their notes and other useful information for their assigned tasks difficult. Some failed to recognize when they were stuck and needed help. In summary, the new developers join large, pre-existing teams as the most inexperienced member, in a situation where they are working with a pre-existing codebase with bugs that predate them with little easily understandable documentation for the newcomer. All of these observations indicate that new software developers transition abruptly into situations very different from the typical project experiences and software engineering course in college.

Begel and Simon (2) make several suggestions for pedagogical changes in CS classes, many of which PL design, implementation, and application courses could incorporate very naturally:

- provide students with a large codebase in which they must fix (injected or real) bugs and write additional features. In a PL course, students could be assigned to add new features or change a language feature in an existing translator, such that they need to understand the representations and algorithms in the existing compiler or interpreter infrastructure before making modifications or extensions.
- include a management component where students who have taken the course previously or the teaching assistant act as a project manager/mentor and teach them about the codebase, giving them cryptic weekly requirements or testing tasks that they need to resolve as a team.
- engage students in a critical reflection on a previous solution to a problem they are to solve. This could be done as a problem-based learning exercise on previous implementations or uses of key PL concepts, reflecting on different solutions to abstraction or garbage collection, for instance.
- engage students in documenting and organizing their notes on tools, code, and processes through the use of various revision control systems, wikis, and other ways. Students can be required to begin their own personal wiki as a freshman with organized notes from various tools that they learn as they progress through the CS major, including PL courses.

2.2 Pre-graduate Students

To begin with a strong start in graduate school in computer science, an undergraduate should carefully select computer science and math courses that provide a strong foundation in both theoretical aspects and practical experience in programming. Programming Language design, implementation, and application is particularly well suited to bringing together many concepts in computer science, including aspects of language specification through regular expressions (and finite automata), context-free grammars (and pushdown automata) and semantic analysis (with attribute grammars), machine architecture, data structures (intermediate representation creation and manipulation and symbol table management), programming languages, algorithms, and domain-specific considerations. Practical programming experience can be gained in different languages as well as participation in a major software engineering task of building or extending a language translator.

Typically with various thought-provoking projects, the PL course can be a good indicator of problem solving abilities, diligence, perseverance, and creativity, all indicative of a good potential graduate student. While undergraduate research experience is a quality often used as a predictor of success in graduate school, it is not something that can be easily incorporated into a course. However, open-ended projects and homework that require creative problem-solving could be included in a PL course. For the pre-graduate student, the PL course can provide more discovery-based learning activities, with both theory and practice in mind and collaborative teams.

2.3 Recruiting and Retaining Students from Under-represented Groups

It is a well known fact that women and minorities (particularly African Americans, Hispanics, and American Indians) are significantly underrepresented in CS&E academic departments, especially at the undergraduate levels(1; 5). This underrepresentation needs to be addressed as it means a loss of opportunity for individuals, a loss of talent to the workforce, and a loss of creativity in shaping the future of society as computing technology becomes more pervasive (5). There have been various theories and some studies about why these numbers continue to decrease. Many of the recommendations for recruitment and retention of these

groups are also known to benefit computer science students in general.

A paper by Cohoon (4), who has many articles on women in computing, recommends methods for increasing female participation in undergraduate computer science. Among the factors that undergraduate women gave for deciding to major in computer science in her qualitative study of 18 large departments in the U.S. were that *computing offers an opportunity to be creative, and they believed that computing fit their personal strengths and abilities*. In another article (3), Cohoon notes that without adequate peer support, women are likely to leave the major at much higher rates than men, and since there is a disproportionate number of men than women in the undergraduate major, the faculty and larger environment need to create the conditions to retain women. While there are many recommendations listed, those that are most relevant to the classroom setting are:

- promoting interaction among classmates
- developing learning communities and other forms of peer support, and
- providing female role models.

All of these goals can be achieved through team projects, with appropriate role models as project leaders and mentors. Peer support can be provided through the management component described in the previous section, as well as peer support through classmate collaborators on team projects. However, the instructor needs to be careful in choosing the groups to ensure that when possible, no women are isolated in a group of all males, to decrease the isolation factor already evident in the academic computer science community. In addition to group projects, in-class group activities can promote interaction among classmates and establishment of peer mentor relationships. In-class activities can range from more in-depth problem-based learning to simple in-class games such as jeopardy for reviewing for exams. Another in-class group activity specific to PL that builds communication skills as well as peer mentoring and interaction include a court trial weighing two languages against each other, requiring students to do some research on the languages and construct sound arguments for them (6).

Varma's (7) study of 40 undergraduate (junior and senior) students' experiences in CS classes, and

interacting with faculty, advisors, teaching advisors, and fellow classmates at a public minority-serving research university revealed a number of pedagogy-related changes that could be made in the classroom. Among those that can be considered when designing a PL course are:

- making the class require creativity and design skills
- showing how new information can be applied to the real world
- requiring that the students do something tangible to see an outcome
- ensuring that courses are not perceived as having to retake them several times to pass
- being aware of and providing support for different backgrounds especially in math
- designing assignments and deadlines with the personal pressures of part-time students in mind, and
- considering learning styles such as team-based and problem-based learning.

These recommendations suggest open-ended projects allowing for creativity to be rewarded beyond just meeting a specification, projects and classroom examples that demonstrate the real world application of the concepts, designing projects and classroom presentations that are adaptable to different mathematical backgrounds (or building in support for different mathematical backgrounds), creating groups of individuals with different mathematical backgrounds to work together to learn from each other, and again careful use of collaborative teams.

3. Summary

A single course cannot successfully achieve too many goals at once. However, many of the recommendations for the diverse kinds of students can be addressed with a few pedagogical considerations. As we discuss the content of PL design, implementation, and application in our computer science curricula, we need to be cognizant of the pedagogy that reaches the diverse set of students we teach. This paper highlights the major recommendations for students transitioning to software engineering positions immediately, going on to graduate school in computer science, and underrepresented in the academic computer science community.

References

- [1] W. Aspray and A. Bernat. Recruitment and Retention of Underrepresented Minority Graduate Students in Computer Science. *Workshop on Recruitment and Retention of Underrepresented Minority Graduate Students in Computer Science*, 2000.
- [2] A. Begel and B. Simon. Struggles of New College Graduates in their First Software Development Job. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2008.
- [3] J. M. Cohoon. Toward Improving Female Retention in the Computer Science Major. *Communications of the ACM*, 2001.
- [4] J. M. Cohoon. Recruiting and Retaining Women in Undergraduate Computing Majors. *SIGCSE Bulletin*, 34(2), 2002.
- [5] J. Cuny and W. Aspray. Recruitment and Retention of Women Graduate Students in Computer Science and Engineering. *Workshop on Recruitment and Retention of Women Graduate Students in Computer Science and Engineering*, 2000.
- [6] L. Pollock. The Supreme Court Case on Java Versus C++. <http://www.eecis.udel.edu/pollock/670/court.html>, 2000.
- [7] R. Varma. Making Computer Science Minority-friendly. *Communications of the ACM*, 49(2), February 2006.