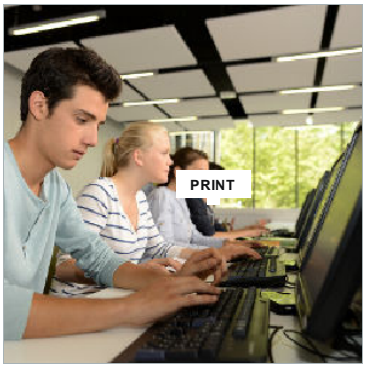
PRACTICE

Undergraduate Software Engineering

By Michael J. Lutz, J. Fernando Naveda, James R. Vallino
Communications of the ACM, Vol. 57 No. 8, Pages 52-58
10.1145/2632361
Comments (1)



In the fall semester of 1996, Rochester Institute of Technology (RIT) launched the first undergraduate software engineering program in the U.S.[9,10] The culmination of five years of planning, development, and review, the program was designed from the outset to prepare graduates for professional positions in commercial and industrial software development.

From an initial class of 15, the ABET-accredited program has grown steadily, Today, the student body numbers more than 400 undergraduates. Co-op students and graduates are employed in organizations large and small, including Microsoft, Google, Apple, and United Technologies, as well as a variety of government agencies. Housed in a separate Department of Software Engineering at RIT, the program has the independence and flexibility necessary to ensure its integrity as it evolves.

Undergraduate Software Engineering: Addressing the Needs of Professional Software Development

Michael J. Lutz
J. Fernando Naveda
James R. Vallino

Department of Software Engineering
Rochester Institute of Technology

Introduction and History

In the fall of 1996, the Rochester Institute of Technology (RIT) launched the first undergraduate software engineering program in the United States[9,10]. The culmination of five years of planning, development, and review, the program was designed from the outset to prepare graduates for professional positions in commercial and industrial software development.

From an initial class of 15, the ABET-accredited program has grown steadily over the intervening years until today the student body numbers over 400 undergraduates. Co-op students and graduates are employed in organizations large and small, including Microsoft, Google, Apple, and United Technologies, as well as a variety of government agencies. Housed in a separate Department of Software Engineering at RIT, the program has the independence and flexibility necessary to ensure its integrity as it evolves over time.

The primary focus of the program is on preparing professional, practicing software engineers. This is illustrated most directly by the required year of co-operative education following two years of foundational coursework. Students alternate terms of formal study with paid professional experience; at the end of the five year program, they have both solid academic preparation and significant practical experience. Our graduates are in high demand, as they are prepared to define, design, develop and deliver quality software systems.

The question remains, of course: Why a specialized software engineering degree? After all, the majority of new industrial hires come from traditional programs in computer science and engineering. The section that follows provides our rationale for striking out in a new direction - our strong belief that, there is a need in industry for entry-level engineers of software, and our conviction that we could provide an educational experience that better prepares students for careers in the software field. Next we discuss the resulting differences between our program and those typical of undergraduate computer science. This leads, in turn, to a presentation of our pedagogical approach, and the state of software engineering in computer science curricula. The final sections discuss our relationship with industry, including comments on the preparation of co-op students and graduates.

Motivation

In the late 1980s, one of us (Lutz) took a two-year industrial leave from RIT. First at GCA/Tropel, a manufacturer of optical metrology products, and later at Eastman Kodak, he led teams developing both embedded systems and application level software. Part of his responsibilities included interviewing, hiring, and mentoring new college graduates, and what he observed during this time was unsettling. By and large these graduates had a solid background in basic computing theory and technology. Many had taken courses in algorithm analysis and theory of computation, and most had some exposure to operating systems, programming language concepts, artificial intelligence, graphics, and compiler design. What they lacked, however, was background necessary to be effective when working on large, complex, industrial quality systems.

In particular, these graduates had few (if any) experiences working as members of a software team, yet this is commonplace industrial practice. Their knowledge of design was often confined to those artifacts of interest to computer scientists - compilers, operating systems, graphics libraries, etc. - yet they had little appreciation of design as an activity in its own right. Most had no experience with version control, much less configuration management. Their knowledge of testing was usually meager, and few had even heard of verification and validation. Finally, they knew little or nothing about the actual processes involved in creating a product beyond rote memorization of the waterfall model.

In conversations with others both in industry and teaching software engineering, it became clear that these problems were pervasive. The question was the old one of science vs. engineering - those whose goal is to grow and expand knowledge vs. those who apply such knowledge to create useful products. Traditionally this is expressed as scientists "build in order to learn" while engineers "learn in order to build." It seemed to many of us that it was an opportune time to apply this distinction to computer science and the engineering of software, just as the difference between physics and the engineering of physical artifacts had emerged in the past.

At the time when we were developing the curriculum, many masters programs in software engineering were already being offered. The prevailing opinion was that undergraduates should pursue computer science degrees, and later enroll in masters programs to complete their education. Given that most computer science graduates go into industry immediately upon graduation, and many may never complete a Master of Science in Software Engineering (MSSE), this approach was problematic for us. Consider the following: One way to teach a new car driver would be to present the theory of the internal combustion engine, the drive train, and the electrical system, then turn over the keys and let the driver take the car for a spin. After running into lamp posts and destroying a few mailboxes, the instructor then says "now you are ready to learn how to drive." From our perspective, this is analogous to the BSCS/MSSE suggestions for preparing software developers.

There must be tradeoffs, of course. Just as a mechanical engineer does not have nearly the depth in physics of a physics major, a software engineer will not have the depth in computer science that a computer science major acquires. However, our argument is that the software engineering knowledge will compensate for the lack of deep scientific

knowledge when it comes to contemporary software development practice. The next section explores the differences between the science of computing and the engineering of software as a way of illuminating the tradeoffs we made.

The Manifesto for Software Engineering Education

In 2001, a group of leading software engineering professionals issued the Manifesto for Agile Software Development[2]. In it they presented a series of tradeoffs between different approaches to software development, such as “responding to change” versus “following a plan.” They conclude by stating that while there is value in all of these approaches, they promote one set - the “agile” approaches - over the other.

Similarly, we do not dismiss the value of traditional computer science approaches; we just value other approaches more in the education of software engineers. Taking our inspiration from the Manifesto, we present the relevant approaches and tradeoffs below. While we recognize that some computer science programs and faculty do incorporate one or more of our approaches, we have found none that do so to the same degree as our program. More details of our program are available on our curriculum flowchart[12].

Horizontal vs. Vertical Curricula. Perhaps the largest difference in philosophy is the trade-off between breadth of engineering knowledge versus the depth of specific technical expertise. The RIT software engineering program is unabashedly based on the former. In our required courses students make several iterations through the entire development lifecycle, from customer requirements to product delivery, with all the activities in between (e.g., formal and informal modeling; architecture and design; testing and quality assurance; planning, estimation and tracking; process and project management). Of course individual courses will focus on specific aspects of the engineer’s professional responsibilities, but by the time of the capstone senior project students have the background they need to take a project from inception to completion.

Indeed, the senior project is itself illustrative of the horizontal approach. Whereas software engineering in many computer science programs is confined to a single term project, for our students the two-term, team-based senior project is the culmination of all that precedes it. Working with real customers, whether industrial, non-profit or internal to RIT, teams are responsible for establishing the project scope, negotiating requirements, designing a solution under constraints (e.g., compatibility with an existing system), performing risk analysis, and creating and enacting an appropriate development plan. The success of this approach is attested to by the many projects that have gone into live operation at project sponsor’s sites.

Teamwork vs. Individual Activity. Many computer science courses emphasize individual competency over teamwork, but working on teams to solve problems is a hallmark of the software engineering program. Indeed, with two exceptions (a course on personal software engineering and one on formal mathematical modeling), all of the software engineering courses incorporate team projects as a significant graded component.

The second year introductory course (taken by software engineering, computer science, and computer engineering majors) promotes teamwork as fundamental to professional practice. Specific roles and responsibilities are highlighted, as are issues of team

cohesion, conflict resolution, and team-based success. This is also the course where we ensure exposure to version control, as this is essential to providing a log of member contributions and to detecting and reconciling conflicting changes to documents and source code.

For that portion of the grade based on team activities (approximately 50%), teams receive grades as a whole. However, we also assess each member in terms of his or her contributions to the team, adjusting each individual's grade based on instructor observations, version control logs, and confidential peer evaluations.

The introductory course is the only one required of computer science and computer engineering majors, but it is merely the first of many for software engineers. We expect students to work in teams of 4-6 as an integral part of their professional education. Through the course of their program, the typical SE students will work on over twenty different teams.

Design and Modeling vs. Programming and Coding. Our students primary exposure to programming per se is in the introductory computer science sequence. While we do discuss programming techniques in both introductory and advanced software engineering courses, this is never the focus. Instead, we view programming competence as providing entry into the field - a basic membership requirement if you will. While teams will program software systems in most of our courses, the focus is on programming as a necessary step on the way to product delivery.

An example may help illustrate this. While we expect students to have an understanding of data structures from their computer science courses, and to grasp the basic concepts of complexity (i.e., "big-O" notation), we rarely require them to build such structures from scratch. Instead, we strongly encourage them to incorporate existing components where possible. The components may be part of the standard environment for languages such as Java or Ruby, or may be provided by third parties (e.g., Ruby gems). Teams are responsible for due diligence to ensure the selected components exhibit certain quality attributes while providing the required functionality, and they must give proper attribution for any components they employ.

Reducing the teaching of programming *per se* gives us room to emphasize more significant issues of modeling and design[14]. The second year introductory course, in addition to the teamwork component discussed previously, is also the one that introduces basic design qualities such as cohesion and coupling, information hiding, designing to an interface rather than an implementation, and abstraction into components delivering well-defined services. Other second year courses for majors provide additional experience with modeling and design.

The modeling course addresses formal based approaches to modeling, exploring, and verifying designs. Using tools such as Alloy[6] and Promela/Spin[5], students learn to express structural and behavioral properties using discrete mathematics, and to use associated tools to verify assertions about overall system properties. In addition, the course provides an overview of data modeling and relational database theory. At the conclusion of the course, students have a better appreciation for the role of rigorous design analysis in software system analysis.

The subsystem design course explicitly addresses design, using design patterns[4] as a vehicle to raise the level of abstraction. Our experience is that by naming common structural and behavioral interactions, and applying this expanded vocabulary in design exercises, students begin to draw away from the implementation details and focus on the higher level component relationships. Later design courses, whether addressing security, concurrency, or web-based systems, can build on this base to discuss design concepts and tradeoffs.

The course on concurrent and distributed system design illustrates another difference from most computer science curricula. Whereas computer science typically introduces these issues in the context of operating systems or database systems, our course is less about the artifacts than concurrent and distributed concepts and issues in their own right. Typically, student teams design, develop and deliver systems other than databases or operating systems where concurrency is a critical design concern, and thus do not view it as the province of specialists. In the age of multi-core computers and cloud computing, this approach has served graduates well.

Disciplined Process vs. Ad-hoc Development. When we were creating the curriculum, the notion of teaching professionalism as encapsulated in a disciplined process was prominent in our thinking. Process is not, as some claim, the be-all-and-end-all of software engineering, but it does provide the frame within which software development takes place. In our curriculum, process is as important a pillar as software design.

This does not mean, however, that we impose one dogmatic approach to process - indeed, we ensure that students are familiar with many process approaches, from strictly planned to the more adaptive agile approaches[3]. Part of being an effective practitioner is to recognize the importance of selecting and adhering to a process appropriate to the project at hand. The benefits of agile approaches for rapidly evolving web systems become significant risks when applied in safety-critical settings (e.g., aircraft fly-by-wire controls).

Part of our process emphasis that is rarely discussed in computer science programs is estimation and tracking. In our first year course on personal software engineering, students estimate and track the effort involved in their in-class activities and longer projects. To prevent “cooking the books,” students are assessed not on how accurate their estimates are, but on their reflections as to why the estimated and actual effort differed. It is such reflective practice that slowly but surely improves students’ estimating ability, which is the foundation for team estimates in later courses.

A Pedagogical Approach

Active learning and team-based project work are the two most prominent characteristics of the pedagogical approach that we use in our software engineering courses[8,13]. Using an active learning pedagogy is certainly not unique to software engineering programs, but having it applied across the curriculum is somewhat unique. We are fortunate that we were able to incorporate support for it in our facilities. We teach almost all of our courses in studio labs, and have replaced significant lecture time with class exercises and team project activities that engage the students in immediate reinforcement of

course concepts. The studio labs, with computers at each seat, provide seamless transitions through lecture, and individual or pair exercises.

Each of our courses has team projects through the entire term with at least 40% of the final grade based on these team activities. To support those activities both during and outside of class times, we provide eleven seven-person teamrooms, each with a whiteboard, desktop computer, and projector. We consider the teamrooms to be extensions of the studio labs. A typical class session might spend half the time in the studio lab moving between lecture and class exercises, and the remainder of the class in the teamrooms. We use the time for exercises in random teams that engage the students with material from that class session, or grouped in their current project teams to do project work. During project work, the instructor directly interacts with each team to gain a better understanding of how well the team is performing both individually and as a team. The team has multiple opportunities to receive project feedback, and design guidance. When designing this pedagogical approach, many of the software engineering faculty remembered the initial period of their industrial experience when much of their instruction in software design came from mentoring by senior engineers. This instructor time with teams enables those interactions.

Over time, we realized that the teams needed even more facility support. The teamrooms were excellent for holding meetings with senior project sponsors, design meetings, and inspections, but they were not adequate for team implementation sessions. To address that, we reconfigured one of our studio labs into the Software Engineering Collaboration Lab with five collaboration areas for six students each. A wall-mounted monitor displays the output from one of four under-table workstations or a student laptop. The workstation monitors are fixed low to the table leaving the airspace of collaboration open. Several industrial visitors commented on the uniqueness of this arrangement, and wanted to recreate it in their own team areas.

Has Anything Changed in Computer Science?

The motivation for creating an undergraduate software engineering program was our perception of a mismatch between the skills that an entry level software developer needed and what was typically provided to students in computer science programs. We believe that the skill set mismatch described twenty years ago still exists in computer science programs. One place to see that is with the recently created Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science (CS2013)[7]. Many computer science programs use these guidelines as their foundation.

The *Principles* that guided the creation of CS2013 specify that “Curricula must … include professional practice (e.g., communication skills, teamwork, ethics) as components of the undergraduate experience. Computer science students must learn to integrate theory and practice, to recognize the importance of abstraction, and to appreciate the value of good engineering design.” One of the expected characteristics of computer science graduates is *Project experience* where “all graduates of computer science programs should have been involved in at least one substantial project. In most cases, this experience will be a software development project, but other experiences are also appropriate in particular circumstances. … Students should have opportunities

to develop their interpersonal communication skills as part of their project experience." Both of these overarching aspects of the guidelines identify a need for software engineering concepts.

Another place where guidance for curricular content in computer science programs exists is in the ABET Criteria for Accrediting Computing Programs[1]. The *Student Outcomes* specified, in the section *Program Criteria for Computer Science and Similarly Named Computing Programs* are stated as:

> *The program must enable students to attain, by the time of graduation:*
>
> *(j) An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the tradeoffs involved in design choices. [CS]*
>
> *(k) An ability to apply design and development principles in the construction of software systems of varying complexity. [CS]*

These two outcomes define a clear need for coverage of design principles and development practices, both of which fall under the software engineering realm. Moreover, we would argue that the modeling and design part of (j) and all of (k) is software engineering.

With the need for software engineering established in the CS2013 guidelines and ABET accreditation requirements, we will now look at what the current curriculum guidelines provide in that area. CS2013 defines 18 Knowledge Areas revolving around technology areas, such as, Architecture and Organization, Graphics and Visualization, Networking and Communication, Operating Systems, and Programming Languages. Only three, Software Development Fundamentals (SDF), Software Engineering (SE), and Social Issues and Professional Practice (SP), contain knowledge that we consider falls into the software engineering realm. Guideline comments identify the SE and SP knowledge areas as specific curricula areas where teamwork and communication soft skills will be learned and practiced. The minimum lecture hours specified for software engineering topics in these three Knowledge Areas are SDF - 10, SE - 28, and SP - 1.

Even though students will have more time on task doing assignments and project work, and may see additional material discussed in elective courses, in our view, these required minimums remain inadequate to develop the full skill set for an entry-level software engineer. This is especially true when you consider that the Software Engineering Knowledge Area, which at 14 pages is the longest non-cross-cutting Knowledge Area in CS2013, identifies 60 Core topics with 69 Learning Outcomes, and 54 Elective topics with 56 Learning Outcomes. The breadth and depth of this Knowledge Area leads to a lament that the authors regularly hear at software engineering education conference sessions. The CS faculty members responsible for software engineering in the curriculum ask "How am I going to fit the core SE topics and the 'soft' teamwork and communication skills in the single software engineering course in our computer science curriculum?" The reality of undergraduate computing education is that the vast majority of students do not go through software engineering curricula where there is time to address this in depth. Instead, they are in computer science or computer engineering programs, and learn their software engineering skills in their one, and often only, software engineering course[15].

Industrial Perspectives on Software Engineering Education

In the May 2013, one author (Vallino) attended a meeting of the Rochester Java Users Group. Instead of hearing a presentation on some aspect of Java technology, Bryan Basham led a general discussion of Software Education/Training/Certification. Basham, who is an active developer and former Java trainer for Sun Microsystems, expressed a concern that there was a mismatch between what was being taught and the skill set that software developers needed. This was the same insight that twenty years prior led us to start developing our software engineering program! The users group session progressed by having the audience list what they remember learning in their undergraduate coursework. This list clearly identified the technology areas that are explored in a traditional computer science degree. Next, the audience described the skills that they felt were needed to be competent at their software development activities. This list covered most elements of our software engineering program and included the need for strong teamwork and communications skills. The experience at this presentation again reinforced that software engineering programs address the needs of professional software development, at least as perceived by an audience of active developers, and that the programs need more visibility because no one in the audience even knew of the existence of undergraduate software engineering.

The effectiveness of our program is quantitatively assessed in our accreditation self-studies. We do have anecdotal comparative indications between Computer Science and Software Engineering. RIT's career services office publishes salary data[11]. Software Engineering students report the highest average hourly co-op and median full-time wages almost every year compared to Computer Science, Computer Engineering, and all the other computing majors at RIT. Our placement rate of undergraduates is over 90% at graduation. In addition, a review of co-op employment evaluations provides other anecdotal evidence of the value of our students' training to their employers. An engineering manager in an aerospace company, which has hired many of our students on co-op and in full-time positions, commented that the students have a strong focus on capturing requirements and system modeling. An engineering vice-president, who has hired students and sponsored senior projects, commented that our graduates match up favorable against some software engineers who have been working for him for five years.

One of our lecturers, Robert Kuehl, who has a 30+ year career developing and managing the development of software systems in consumer and commercial imaging, gave this assessment of the skills preparation that our students receive:

> *In a generalization, industry wants:*
>
> - *Professionalism. Individuals who act professionally, communicate effectively verbally and in writing, and who work effectively in diverse teams.*
> - *Execution competence. Professionals who know how to elicit and specify good requirements, who can transition requirements into designs that fulfill requirements, who can productively write good code, debug code, and test code. They want professionals who effectively select and execute*

software development methodologies and tools to manage projects that are consistently delivered on time and within budget.

- *Technical knowledge and expertise. Professionals who are on top of current technology, who have sound knowledge of computing principles, techniques, and algorithms, and who can innovate.*

The computer science curricula helps students unquestionably gain computing technical knowledge and expertise. The software engineering curriculum provides similar technical grounding but integrates other course work to teach professionalism, and to acquire the execution competence that comes with it. Courses cover all aspects of the software development life-cycle in depth via course projects which emphasize learning by doing, teamwork, and communication in addition to the technical aspects of the projects.

As a result in my experience, software engineering graduates are generally better prepared for jobs in industry that require the development and deployment of quality software.

Jeffrey Lasky, Professor, Information Technology, served as an RIT Professor in Residence at Excellus, a local Blue Cross/Blue Shield health insurance provider. Conversations with Prof. Lasky first tipped us off to some of the distinctions generated by the software engineering curriculum.

The RIT/Excellus BlueCross BlueShield co-op program began in Fall 2002. The program was co-managed by the Director, Excellus Architecture and Integration Group, and an RIT Professor-in-Residence. The co-op program was open to all RIT undergraduate students majoring in a computing discipline.

In 2004, a team of six students, two each from computer science, information technology, and software engineering, were assigned to work on a subsystem for a high priority, system development project. The composition of the team was unplanned but serendipitous. The students quickly realized that their respective skill set strengths clustered around a core area of their degree program: programming (CS), database and Web (IT), and design (SE).

While all topics proved to be of interest, the software engineering students' use and explanations of software design patterns gathered the most attention from the other students, who quickly noticed that the SE students thought differently about software systems development. Specific patterns became part of and often guided team dialogues; the CS and IT students were enthusiastic about the role and value of formal abstractions in software design. The supervising Excellus software architects had similar reactions to design patterns, and copies of the classic book, Design Patterns: Elements of Reusable Object-Oriented Software, started to appear on their desks, and thereafter, on the desks of many Excellus software developers.

Professor Lasky's characterization of the student strengths in the various degrees in our college has been echoed by colleagues at other institutions. The SE students ask questions about components, architecture, and interactions between the components, preferring a higher-level and more abstract model-driven discussion. The CS and IT

students tend to ask for examples of working code and begin understanding the system from the bottom up. The CS students are great at coding but generally lack skills in design, and concern for quality attributes. With our curricular balance between design and process, the SE students have a broader range of coding skills with some students not interested in doing much coding. When we have surveyed our students, two-thirds preferred the design and implementation side, and the rest were more interested in the process side, e.g. requirements, process improvement, and software quality assurance.

Conclusion

Twenty years ago when we started creating the first undergraduate software engineering program in the US, we gambled that if we built it, they would come, which includes an attraction for both students and employers alike. Our track record of continual program growth and over 90% placement of graduates demonstrates that the gamble paid off. We believe that our software engineering program, which concentrates on engineering design, software product development, teamwork, and communication, provides students who seek a career in software development with a set of skills better tailored to what is needed to excel not only as an entry-level software engineer but also for growth throughout their career.

References

1. ABET Computing Accreditation Commission. Criteria for Accrediting Computing Programs, (2013) http://www.abet.org/uploadedFiles/Accreditation/Accreditation_Process/Accreditation_Documents/Current/C001%2014-15%20CAC%20Criteria%2010-26-13.pdf. Accessed May 2, 2014.

2. Beck, K. et al. Manifesto for Agile Software Development. http://www.agilemanifesto.org/ (2001) Accessed May 2, 2014

3. Boehm, B.W. and Turner, R. *Balancing agility and discipline: a guide for the perplexed*. Addison-Wesley, Boston, 2004.

4. Gamma, E. et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, (1995).

5. Holzmann, G.J. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Educational Publishers, New Jersey, (2011).

6. Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, (2012).

7. Joint Task Force on Computing Curricula. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. (2013).

8. Ludi, S., Natarajan, S., and Reichlmayr, T. An introductory software engineering course that facilitates active learning. In *Proceedings of SIGCSE Technical Symposium on Computer Science Education*. (2005), 302.

9. Lutz, M.J. and Naveda, J.F. The road less traveled: a baccalaureate degree in software engineering. In *Proceedings of Conference on Software Engineering Education and Training*. (1997).

10. Naveda, F. and Lutz, M. Crafting a baccalaureate program in software engineering. In Proceedings of *Twenty Eighth SIGCSE Technical Symposium on Computer Science Education*. San Jose, CA, (1997).

11. RIT Office of Cooperative Education and Career Services. Students - Salary Data | Office of Co-op and Career Services. (2013). http://www.rit.edu/emcs/oce/students/salary. Accessed May 2, 2014

12. RIT Department of Software Engineering. Undergraduate Software Engineering Curriculum. (2013) http://www.se.rit.edu/pagefiles/documents/VSEN%20Flowchart%20v6.6%202013-09-08.pdf. Accessed June 2, 2014.

13. Vallino, J. Design patterns - evolving from passive to active learning. In *Proceedings of Frontiers in Education Conference*. (2003).

14. Vallino, J. If you're not modeling, you're just programming: modeling throughout an undergraduate software engineering program. In *Proceedings of the 2006 International Conference on Models in Software Engineering*, (2006), 291–300.

15. Vallino, J. What should students learn in their first (and often only) software engineering course? In *Proceedings of the Conference on Software Engineering Education and Training,* (2013), 335–337.