📄 PDF

Undergraduate Software Engineering

Addressing the Needs of Professional Software Development

Michael J. Lutz, J. Fernando Naveda, and James R. Vallino
Department of Software Engineering, Rochester Institute of Technology

In the fall semester of 1996 RIT (Rochester Institute of Technology) 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. 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.

Its primary focus 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. These 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 prepared students for careers in the software field. Then the article looks at the differences between the RIT 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 the RIT program's relationship with industry and the preparation of co-op students and graduates.

Motivation

In the late 1980s, one of this article's authors (Lutz) took a two-year industrial leave from RIT: first at GCA/Tropel, a manufacturer of optical metrology products; and later at Eastman Kodak, where he led teams developing 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 the background necessary to be effective when working on large, complex, industrial-quality systems.

In particular, these graduates had little (if any) experience working as members of a software team, yet this is common 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.

Conversations with others, both in industry and software engineering education, indicated that these problems were pervasive. The issue 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." This seemed 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 we were developing the software engineering curriculum at RIT, many master's programs in software engineering were already being offered. The prevailing opinion was that undergraduates should pursue computer science degrees and later enroll in master's programs to complete their education. Given that most computer science graduates go into industry immediately upon graduation, and many may never complete an MSSE (master of science in software engineering), this approach was problematic. Consider the following: one way to teach a new 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 the new driver has run into some lamp posts and destroyed 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 approach to educating 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. Our argument, however, 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 these tradeoffs.

The Manifesto for Software Engineering Education

In 2001 a group of leading software engineering professionals issued the Manifesto for Agile Software Development.[2] They presented a series of tradeoffs among different approaches to software development, such as "responding to change" versus "following a plan." They concluded 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 agile manifesto, we present the relevant approaches and tradeoffs of our software engineering program at RIT. While recognizing 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 the RIT program are available on the curriculum flowchart.[12]

Horizontal vs. Vertical Curricula

Perhaps the largest difference in philosophy is the tradeoff between breadth of engineering knowledge (horizontal) versus the depth of specific technical expertise (vertical). The RIT software engineering program is unabashedly based on the former. In required courses students make several iterations through the entire development life cycle, 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 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 illustrative of the horizontal approach. Whereas software engineering in many computer science programs is confined to a single term project, for RIT's software engineering students the two-term, team-based senior project is the culmination of all that precedes it. Working with real customers, whether industrial, nonprofit, 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 sponsors' 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 RIT's 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 significant graded components.

The second-year introductory course (for 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 that ensures 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 percent), teams receive grades as a whole. Each team member is also assessed in terms of his or her

contributions to the team, based on instructor observations, version–control logs, and confidential peer evaluations.

The introductory course is the only such course required of computer science and computer engineering majors, but it is merely the first of many for software engineers. Students are expected to work in teams of four to six members as an integral part of their professional education. Through the course of the program, the typical software engineering student will work on more than 20 different teams.

Design and Modeling vs. Programming and Coding

The RIT software engineering student's primary exposure to programming *per se* is in the introductory computer science sequence. While programming techniques are discussed 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 students are expected 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), they are rarely required 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 they may be provided by third parties (e.g., RubyGems). 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* leaves 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 multicore computers and cloud computing, this approach has served graduates well.

Disciplined Process vs. Ad-hoc Development

In creating the software engineering curriculum at RIT, 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 the process emphasis that is rarely discussed in computer science programs is estimation and tracking. In the 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 used in RIT's 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. Fortunately, we were able to incorporate support for it in our facilities. Almost all of our courses are taught in studio labs, and we have replaced a significant amount of 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 course has team projects through the entire term that account for at least 40 percent of the final grade. To support the team activities both during and outside of class times, there are 11 seven-person team rooms, each with a whiteboard, desktop computer, and projector. The team rooms are 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 team rooms.

Students use this time in the team rooms either grouped in random teams to engage in exercises using material from that class session, or grouped in their current project teams to do specific project work. During this time, the instructor interacts directly with each team to gain a better understanding of how well both the team and its individual members are performing. The team has multiple opportunities to receive project feedback and design guidance.

When designing this pedagogical approach, many of the software engineering faculty members 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 encourages those interactions.

Over time, we realized that the teams needed even more facility support. The team rooms 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 shortcoming, we reconfigured one of our studio labs into the Software Engineering Collaboration Lab, containing five collaboration areas accommodating 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 visitors who work in the industry voiced their appreciation for this unique arrangement, wishing to re-create 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 20 years ago still exists. One place to see that is in *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* (CS2013),[7] used as the foundation for many computer science programs.

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 titled 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) are software engineering.

With the need for software engineering established in the CS2013 guidelines and ABET accreditation requirements, let's look at what the current curriculum guidelines provide in that area. CS2013 defines 18 knowledge areas revolving around technology such as architecture and organization, graphics and visualization, networking and communication, operating systems, and programming languages. Only three—SDF (software development fundamentals), SE (software engineering), and SP (social issues and professional practice)—fall within 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 10 in SDF, 28 in SE, and one in SP.

Even though students will have more time on task doing assignments and project work, and may see additional material discussed in elective courses, these minimums are inadequate for developing the full skill set for an entry-level software engineer. This is especially true when you consider that the SE 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 heard regularly 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 they learn their software engineering skills in their one, and often only, software engineering course.[15]

Industrial Perspectives on Software Engineering Education

In May 2013, one of this article's authors (Vallino) attended a meeting of the Rochester Java Users Group. Instead of hearing a presentation on some aspect of Java technology, the group had a general discussion of software education/training/certification led by Bryan Basham,

who is an active developer and former Java trainer for Sun Microsystems. He expressed 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 20 years prior led us to start developing our software engineering program at RIT.

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 RIT's software engineering program and included the need for strong teamwork and communications skills. This experience reinforced the idea 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 RIT's software engineering program is quantitatively assessed in accreditation self-studies. We do have anecdotal comparative indications between computer science and software engineering. RIT's career services office publishes salary data,[11] and software engineering students report the highest average hourly co-op and median full-time wages almost every year compared with computer science, computer engineering, and all the other computing majors at RIT. The placement rate of software engineering undergraduates is more than 90 percent at graduation.

In addition, a review of co-op employment evaluations also provides 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 favorably with some software engineers who have five years of experience at the company.

One of our lecturers, Robert Kuehl, who has 30-plus years in a 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 curriculum helps students unquestionably gain computing technical knowledge and expertise. The software engineering curriculum provides similar technical grounding but integrates other coursework 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 that 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 of information technology, served as an RIT Professor in Residence at Excellus, a local Blue Cross/Blue Shield health insurance provider. Conversations with Lasky first tipped us off to some of the distinctions generated by the software engineering curriculum.

"The RIT/Excellus Blue Cross/Blue Shield 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 the curricular balance between design and process in RIT's software engineering program, students have a broader range of coding skills, with some students not interested in doing much coding at all. In student surveys, 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 RIT started creating the first undergraduate software engineering program in the U.S., we gambled that if we built it, they would come—meaning both students and employers. The program's track record of growth and more than 90 percent placement of graduates demonstrates that the gamble paid off. The concentration 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 not only to excel as entry-level software engineers, but also to prepare them for growth throughout their career.

References

1. ABET Computing Accreditation Commission. 2013. Criteria for accrediting computing programs; http://www.abet.org/uploadedFiles/Accreditation/Accreditation_Process/Accreditation_Documents/Current/C001%2014-15%20CAC%20Criteria%2010-26-13.pdf.

2. Beck, K., et al. 2001. Manifesto for agile software development. http://www.agilemanifesto.org/.

3. Boehm, B.W., Turner, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley.

4. Gamma, E., et al. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading, MA: Addison-Wesley.

5. Holzmann, G.J. 2011. *The SPIN Model Checker: Primer and Reference Manual*. Upper Saddle River, NJ: Addison-Wesley Educational Publishers.

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

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

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

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

10. Naveda, F., Lutz, M. 1997. Crafting a baccalaureate program in software engineering. In Proceedings of the 28th *SIGCSE Technical Symposium on Computer Science Education*.

11. Rochester Institute of Technology, Office of Cooperative Education and Career Services. 2013. Salary | Program Overview; http://www.rit.edu/emcs/oce/students/salary.

12. Rochester Institute of Technology, Department of Software Engineering. 2013. Undergraduate curriculum;

http://www.se.rit.edu/pagefiles/documents/VSEN%20Flowchart%20v6.6%202013-09-08.pdf.

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

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

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

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

Michael Lutz (mjlvse@rit.edu) has been on the RIT faculty since 1976. In the 1990s he initiated and led the effort to develop the first baccalaureate software engineering program in the United States. His professional interests include software engineering education, software design, and mathematical modeling of software.

J. Fernando Naveda (F.Naveda@rit.edu) is co-founder of RIT's Department of Software Engineering, which he chaired from 2001 through 2010. His interests are in curriculum development and a broad range of software engineering academic topics. He has been at RIT for 21 years, where he currently serves in the Academic Affairs Office.

James Vallino (J.Vallino@se.rit.edu) has been associated with RIT's Department of Software Engineering since he started at RIT in 1997. He has been the chair of the department since 2010. Prior to RIT, he had 16 years of industrial experience at AT&T Bell Laboratories, AVL, and Siemens Corporate Research. His interests are in the development of undergraduate software engineering curricula with an emphasis on software design and modeling.