The Road We've Traveled: 12 Years of Undergraduate Software Engineering at the Rochester Institute of Technology

J. Fernando Naveda
Thomas J. Reichlmayr

Michael J. Lutz

James R. Vallino
Stephanie A. Ludi

Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA

Abstract

In 1996, the Rochester Institute of Technology launched the first undergraduate software engineering program in North America. This paper briefly reviews the development of the program, and describes the program's evolution up to the present. We illuminate both the constant aspects of the program – what we believe we got right – as well as the changes made in light of pedagogical, technological and disciplinary advances. We conclude by considering the current and future challenges for undergraduate software engineering education both at RIT and elsewhere.

1. Introduction

In 1996, RIT established the first undergraduate software engineering program in North America. This was consistent with RIT's long tradition of curricular innovation, as exemplified by other programs such as Microelectronics Engineering (1982) and Information Technology (1992). In the intervening 12 years, over 260 students have graduated from the program and begun careers in segments of the software industry in areas such as, gaming, imaging, embedded systems, Internet applications, and many areas in between. The faculty has gone from two full-time positions to eight full-time professors, one lecturer, and several adjuncts.

The program's development was first described in "The Road Less Traveled: A Baccalaureate Degree in Software Engineering"[1]; with this paper we bring the history up to date, discus the road we've traveled, changes made, and how the program changed along the way, and consider the path that lies in front of us.

2. History of the program

While software engineering concepts had been included in some computing courses at RIT, by the early 90's many of us believed a full baccalaureate program was appropriate. Whereas computer science properly focused on the theory and science of computing, and computer engineering emphasized digital systems hardware, there was no program that provided sustained instruction in the engineering of software. That is, no program could claim to prepare graduates with "the knowledge and skills needed to design and create software products that satisfy customers and users[1]." In our view, masters level programs came too late – there was no reason why the fundamental concepts could not be part of undergraduate education.

Thus, in 1993 a faculty task force was formed to develop an undergraduate software engineering program. The task force comprised four members each from computer science and the college of engineering, appointed by the respective college deans, as well as a highly respected senior engineer as facilitator. The task force quickly agreed that the program would satisfy the following constraints:

1. A year of co-operative education; this is a hallmark of RIT's technical curricula.
2. Adherence to the general engineering and program-specific accreditation criteria of ABET's EAC[2].
3. Ease of transfer among software engineering, computer engineering, and computer science during the first two years of study.

What the task force could not agree upon, however, was the specific structure of the curriculum. While the engineering faculty saw this as a way to enhance software in engineering, the computer scientists focused on the engineering of software *per se*. To resolve these differences, representatives from software intensive industries were invited to campus. After discussing both approaches to the program, these advisors recommended a program aligned to the computer science vision.

With this deadlock broken, the curriculum quickly took form, and was approved by RIT and the State of New York Education Department. The first freshman class enrolled in the fall of 1996; administratively, the program's chair reported to both the College of Engineering and the College of Applied Science and Technology, which housed computer science. When the B. Thomas Golisano College of Computing and Information Sciences was created in 2001, the software engineering department, along with computer science and information technology, was transferred to this new college.

3. Main curriculum themes

Our goal was to provide a coherent program of study, not simply a set of courses each encapsulating a discrete piece of the discipline. To help ensure this, we identified seven themes that would pervade the entire program:

Design**:** Designing solutions to customer problems is at the heart of any engineering activity. Multiple courses would concentrate on design of software systems, including design principles and patterns, software architecture, and design evaluation.

Process**:** Coursework would teach and require the application of defined and managed processes as part of software development projects.

Evolution and maintenance**:** Few software systems are created from scratch; in acknowledgement of this, class projects would include experience in maintaining and enhancing existing systems.

Complexity management**:** Graduates need to understand the sources of complexity, as well as tools and techniques to reduce or mitigate system complexity.

Standards**:** Students must understand the role of standards, whether de facto or de jure, as tools to increase the likelihood of project success.

Team-based development**:** Students must be given ample opportunity to practice and develop team skills. These skills, taught in early courses, must be constantly reinforced by team projects in later coursework.

Professionalism**:** Courses must teach the skills, habits, and attitudes characterizing professional engineering practice, and we must hold students accountable for producing professional quality work.

4. Original curriculum architecture

Our original program comprised courses in seven distinct areas: liberal arts, mathematics and science, computer science, engineering, application domains, cooperative education, and one (1) free elective. In accounting for courses, bear in mind that RIT is on the quarter system, and thus the typical student takes four four-hour classes per 10 week quarter, for a total of 12 courses per academic year.

The liberal arts encompass the core and elective study in the humanities and social sciences required of all RIT engineering graduates. For mathematics and science, we followed the lead of other engineering programs, and required a year each of calculus and physics, as well as a course each in differential equations, probability and statistics, and basic chemistry. In support of software engineering, we also required a two course sequence in discrete mathematics.

We viewed computer science as the core science of software engineering, much as physics is the core science of electrical and mechanical engineering. In recognition of this, we required the basic computer science sequence, CS1 to CS4, which covers algorithms, data structures, and which imparts basic programming skills. In addition, our students took computer science courses in professional communications, scientific programming, and programming language concepts.

The heart of the program, of course, consists of courses in traditional and software engineering. Table 1 summarizes the initial engineering component.

Table 1. Original engineering component

Traditional	**Software**
Assembly language	Intro. to software engineering
Digital systems	Software subsystems
Computer architecture	Software architecture
Human factors	Formal methods
	Requirements & specification
	Senior project (I and II)
	Design elective (2)
	Process elective (2)
	Unrestricted SE elective (1)

The incorporation of application domains was a unique aspect of our curriculum, based on our perspective that software engineering as a profession is not performed in a vacuum. Instead, the practices and principles of software engineering are applied within particular application domains (e.g., business, avionics, and entertainment). Thus each student was required to take a cohesive cluster of three courses to impart basic knowledge in one of several predefined application domains. A student could propose a custom domain, as long as it met the same standards for cohesiveness and focus as those defined by the faculty.

Students showed their ability to work in different application domains as part of the two-quarter senior capstone project. Teams of 4-6 students worked on projects proposed by commercial and non-profit sponsors or other units at RIT. Most projects were in unfamiliar domains, and required the team to get a baseline understanding in order to create and execute a project plan. Projects include transit system safety design, emergency medical information services, photo kiosks, and military radio configuration [3].

Finally, all students had to complete five quarters of co-operative education. Co-op has been a mainstay of the RIT approach to education since 1914; under this system, upper-division students alternate between academic quarters on campus and paid junior staff work in industry. In our experience co-op greatly enhances our upper-division courses, as students bring to their classes a level of maturity, expertise, and general "savvy" that is impossible to create in the classroom alone.

5. A look at the engineering coursework

As noted in the previous section, students took a combination of traditional and software specific engineering courses. The assembly language course, in combination with digital systems and computer architecture, helped ensure our students had an appreciation for what happens "under the hood," while enabling them to work on projects right at the hardware/software boundary. Human factors was included to give students experience working with other engineers (in this case, industrial engineers), while exposing them to key issues in interface design.

As one would expect, the bulk of the engineering courses focused on software engineering. The introductory course was required for all computer science, computer engineering, and software engineering majors; for the first two groups, this was the only software engineering required. For software engineering majors, on the other hand, this was the entry point for further study in the discipline. As a consequence, the course was broad, not deep, and required teams to create a modest size application in several iterations, following a prescribed process with defined deliverables. Besides the focus on process-centered teamwork, the course introduced basic design concepts (e.g., cohesion, coupling, separation of concerns), unit and system testing, and contemporary tools such as an IDE and version control. Students were then expected to apply these concepts and use the tools in their later coursework.

The software subsystems course was the first one devoted specifically to design. It is here that student teams delve deeper into the principles of design, assessment of existing and proposed designs, and the application of design patterns to common design problems. The goal was to raise the level at which students view a system; that is, we wanted them to be comfortable working with abstractions. This seems to have worked, as attested to by one of our co-op employers who said "CS students want to see the code; SE students want to see the overall design – they ask questions about components, patterns, and interactions."

The remainder of the design sequence built on this base by narrowing the perspective to specific types of problems. In particular, students had to select two of three design electives addressing issues in concurrent, distributed, and information systems design, and all students took the software architecture course.

In a similar vein, students were required to take a course on requirements and specification, so that they could recognize good and bad requirements, as well as participate in requirements elicitation. This course was coupled with two electives from process-focused courses in software process models, software metrics, and software verification and validation.

The last required course was formal methods (mathematical modeling of systems). Over the years we've employed a variety of modeling methods, most recently Alloy[4] from MIT. What has not changed, however, is the emphasis on applying mathematics to precisely capture design decisions and then using mathematics to deduce system properties.

Finally, as noted previously, all students participated in a two-quarter senior project.

6. Continuous improvement

Every accredited engineering program must have an assessment and continuous improvement plan in place. In our case, the key components of the plan are yearly meetings with an Industrial Advisory Board (IAB), quarterly assessments of course outcomes vis-à-vis ABET criteria, and an annual retreat to review assessment data and plan any changes.

Our annual IAB meeting provides an opportunity for us to solicit advice on any planned changes, request information on our graduates' performance, and seek guidance as to emerging trends we should incorporate in the curriculum. The IAB membership is purposefully broad so that we don't overemphasize issues related to a particular application domain.

Assessment data is gleaned from all required software engineering courses by encoding the level of achievement, by both individuals and teams, on homework, on-line discussions, projects, exams, and in-class activities. These assessments point the way towards systemic changes across the curriculum. In the following paragraphs, we sketch examples of changes we've made to pedagogy and content as a result of our assessments.

Over the years we have incorporated "active learning" strategies in many of our courses. There is ample evidence that such active strategies result in better learning than traditional methods based on lecture alone[5], and the faculty has led the effort to incorporate such approaches into the software engineering curriculum [6, 7, 8].

One result of this change is that we no longer deliver courses in the traditional lecture-lab format. Instead, we employ a studio lab format, where class time is devoted to a combination of lecturing, short group activities, and longer team projects. Our facilities feature studio classrooms and eleven team break-out rooms. We are also investigating blended learning[8),], wiki's for communication and project documentation, and teleconferencing with remote project sponsors.

Originally we offered little instruction in project management, believing that newly hired engineers are unlikely to be faced with such issues. This resulted, however, in poor planning, estimating, and tracking during senior projects. We also noticed a rise in the number of students taking industrial engineering as their application domain; on closer inspection we learned that the students were using this domain to learn project management. Seniors were passing down the word that "you would be very wise to get some project management experience before starting senior projects." We addressed this by changing the existing process course into one titled Software Process and Project Management, which both reviews software development processes and covers fundamental software project management techniques.

Our emphasis on teamwork has led to a difficulty in assessing individual skill and talent, which is itself a prerequisite for effective team participation. Thus all software engineering students now take a Personal Software Engineering course in the second year. The goal of this course is to enhance and assess each student's individual technical abilities prior to the team-based courses that follow.

We also fell short in our commitment to the evolution and maintenance theme. This was brought home forcefully by the following comment on our alumni survey:

> *"In the situation where I have to design new pieces of code for an existing project, RIT has prepared me well. I am constantly creating classes under the guidelines of design patterns, low coupling, high cohesion, etc. I find that I write effective and maintainable code. However, in the situation where I have to maintain existing code, where no features are added, then I wasn't prepared well. I have been learning about refactoring on my own. It would be nice if RIT had a class about what to do in the situation where you work with old code."*

In response, we updated our second year course, Engineering of Software Subsystems course, to include analysis and refactoring of an existing system. The legacy code base used is the deliverable for some team's project in the previous year's Introduction to Software Engineering; it is gratifying for us to see students learn valuable maintenance lessons from the products others have created.

Finally, a self-study revealed that though students can master discrete mathematics, that knowledge quickly fades after taking Formal Methods. We also learned that the students' view of a model was limited to a collection of boxes or symbols connected with arrows and lines that mean something. To partly address these shortcomings, we now require a third-year Concurrent Systems course where students get a generous exposure to issues of modeling and further use of formal methods.

7. Our Program Today

Our current program has the following in common with the initial offering:

- Basic liberal arts (humanities and social science).
- CS 1, 2, and 3.
- One year of calculus and two courses in discrete mathematics.
- Professional communications (now offered by liberal arts).
- The application domain requirement and co-op.
- Software engineering: introductory course, software subsystems, software architecture, formal methods, and software requirements & specification.

Changes to the program include timing of course offerings, new required courses, and more flexible electives. Key aspects of these changes are highlighted below.

We moved the formal methods course earlier in the sequence, exchanging its place with software architecture, based on our observation that only those who had co-op experience really appreciated architectural issues. We require completion of co-op and both the software architecture and requirements and specifications courses prior to enrolling in senior project. Finally, we moved discrete mathematics to the first year of the program to better prepare students for the initial software engineering courses.

While the basic liberal arts requirements have not changed, we have stipulated that every student must take approved courses in economics and ethics. Every engineer should be aware of basic economics concepts, and the same can be said for an exposure to ethical issues in the profession. Approved courses are not necessarily offered by liberal arts – for example, industrial engineering offers engineering economics.

With respect to mathematics and science, we replaced differential equations with a student selected elective. We also removed the requirement for a full year of physics; we only require one physics course coupled with a full year sequence in a lab science (remaining physics, chemistry, or biology); this gives students the opportunity to take application domains such as bioinformatics.

An RIT mandate required us to have at least three free electives in the program which led us to reduce the computer engineering to one custom course. We also replaced the human factors course with a software engineering course more targeted to the engineering of human computer interfaces.

The computer science requirements were altered by exchanging programming language concepts for an introduction to computer science theory. This allows us to proceed at a faster pace in the formal methods course, as well as the design courses using modeling techniques such as state machines.

Finally, the overall engineering component was changed by adding the personal software engineering course mentioned previously, by requiring a software process and project management, as well as concurrent system design, and by opening the engineering electives to allow any engineering course – not necessarily from software engineering – for which the student has the necessary prerequisites.

8. Challenges for the future

Our program has evolved and will continue to evolve in response to increasing competition, industrial needs, globalization, and student demographics. In the absence of competition, during the first 7 years of the program our freshman classes steadily grew 10% to 15% each year. Table 1 shows the growth of our program. Similar to many computing programs, we have experienced declining enrollments. We believe that this decline is primarily due to two factors: increasing competition from other software engineering programs, and the perception that software engineers have no future in the US.

Competition from other universities is a mixed blessing. On the one hand, we are delighted to see others following in our footsteps and that students have an array of options. On the other hand, we must work harder to recruit each student who enrolls. We can continue to attract qualified students by constantly assessing our program and keeping it at the leading edge of software engineering education. Our connection to industry through co-op contacts, senior projects, and the IAB is an advantage in this regard.

Table 1. - Program admissions, graduations, and total enrollment

	Admit	Total		Admit	Graduate	Total
1996	24	24	2001	74	12	240
1997	25	45	2002	83	19	250
1998	55	100	2003	104	25	290
1999	74	160	2004	109	27	310
2000	84	210	2005	75	39	340
			2006	64	45	380
			2007	81	42	420
			2008	58	57	350

Globalization is both a threat and an opportunity. By requiring students to use their communications skills in every class, via written documents and project presentations, we ensure they are prepared. In addition, many senior projects are decentralized, requiring interaction with a remote project sponsor.

Demographics are an issue prominent in the minds of every university today. In the recent past, the numbers of students pursuing undergraduate computing degrees has been on the decline [9]; the dropoff has been especially precipitous among women and minorities even as the outlook for such students had improved[10]. This participation gap exists in our program, where only 6.5% of our students are women, and where the overwhelming majority of students are Caucasian. Overall, the demand for computing professionals, including software engineers, is expected to remain strong for years to come[11]; if we are to respond to this demand, we must reach out to communities we currently do not serve.

One approach is to provide avenues for women and minorities to work in engineering. One of us (Ludi) has worked with other female engineering faculty to create activities throughout the year that will attract girls to engineering, including software engineering. During a yearly three-day, theme-based event, teams of girls from middle-school use Lego Mindstorms® robots to explore the design, development, and testing of an engineered solution. Ludi also facilitates a software engineering themed exhibit using the Lego Mindstroms® robots as part of an annual career fair for local Girl Scouts.

Students with disabilities can find difficulty in computing courses due to accessibility issues with equipment and curricula. General resources, including mentoring and universal design strategies for instructors, provide support for students with disabilities who wish to pursue STEM degrees, including software engineering [12]. In addition outreach projects to promote software engineering have been developed locally to enable students with visual impairments and their parents to explore the field in an engaging and accessible manner [13].

However outreach must include teachers. As part of a new NSF grant, Ludi and Reichlmayr will be conducting workshops for educators to enable them to maximize accessible instruction for students with visual impairments [14].

9. Reflections on the trip so far

Twelve years ago, the new software engineering program was a source of contention between colleges, departments, faculty and students. In addition to the conflict with engineering faculty during the program's development, we also found many of our computer science colleagues were nervous – they thought software engineering would compete for the same pool of students. We have seen that the recruiting pool expanded, and none of the programs has suffered from the competition. As a result, our current relations with both engineering and computer science are excellent, and software engineering has garnered respect for the quality of the program and its graduates.

Today, software engineering is a staple of RIT's academic portfolio. The real-time and embedded system course sequence is jointly sponsored by software engineering and computer engineering, and we have run senior projects that included students and faculty from engineering (computer, mechanical, industrial, electrical), imaging science, print media, and public policy. Students clearly understand how software engineers are different from computer scientists, programmers, and computer engineers. As a result, our program has gained recognition nationally well beyond RIT and into the international community.

We have seen our department grow and our program emulated elsewhere. That is rewarding in its own right, but the final verdict on our work is the success of our graduates. Nothing is more pleasing than knowing that our alumni are moving along with their careers, and that our seniors often secure jobs months before graduation. That is the capstone to a project we started 15 years ago when we set out to take the road less traveled.

10. References

[1] Lutz, M. J. and Naveda, J. F., "The Road Less Traveled: A Baccalaureate Degree in Software Engineering", *Proceedings 28th SIGCSE Technical Symposium on Computer Science Education*, 1997.

[2] ABET, http://abet.org/.

[3] Department of Software Engineering, RIT, http://www.se.rit.edu/?q=node/senior_projects.

[4] Jackson, D., *Software Abstractions*, MIT Press, 2006.

[5] Prince, M., "Does Active Learning Work? A Review of the Research", *Journal of Engineering Education*, v93 n3, July 2004.

[6] Vallino, J., "Design Patterns: Evolving from Passive to Active Learning", *Proceedings of the Frontiers in Education Conference*. Boulder, CO. November 2003.

[7] Ludi, S., Reichlmayr, T., and Natarajan, S., "An Introductory Software Engineering Course That Facilitates Active Learning", *Proceedings of SIGCSE Conference*, St. Louis, MO. February, 2005.

[8] Reichlmayr, T., "Enhancing the Student Team Experience with Blended Learning Techniques", *Proceedings of Frontiers in Education Conference*. Indianapolis, IN. October 2005.

[9] Chabrow, E., "By the Book", *Information Week*, August, 16, 2004; Available: http://www.informationweek.com/story/showArticle.jhtml?articleID=29100069&tid=13692

[10] *Land of Plenty Diversity as America's Competitive Edge in Science, Engineering and Technology: Report of the Congressional Commission on the Advancement of Women and Minorities in Science, Engineering and Technology Development September 2000*; available: http://www.nsf.gov/pubs/2000/cawmset0409/cawmset_0409.pdf

[11] Bureau of Labor Statistics, *Occupational Outlook Handbook 2008-09*, "Computer Software Engineers". Available: http://www.bls.gov/oco/ocos267.htm

[12] University of Washington DO-IT Center: http://www.washington.edu/doit/

[13] Ludi, S., and Reichlmayr, T., "Developing Inclusive Outreach Activities for Students with Visual Impairments", *Proceedings of SIGCSE Conference,* Portland, OR. February, 2008.

[14] Ludi, S., and Reichlmayr, T., "BPC-DP Project ACE: Accessible Computing Education for Visually Impaired Students," NSF Award # 0837493.