METHODS OF TEACHING LEADERSHIP TO UNDERGRADUATE STUDENTS IN COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Steve Chenoweth

Rose-Hulman Institute of Technology; Email: chenoweth@rose-hulman.edu

1. INTRODUCTION

Rose-Hulman Institute of Technology (RHIT) is atypical in requiring a course in requirements engineering and a course in project management for undergraduate computer science (CS) majors. These courses also are required for software engineering (SE) majors, who further must take a course in software architecture and design. All three required courses are generally in the area of management and technical leadership, and it is unusual for them to be a required part of bachelors-level curricula. The courses help students understand overall software processes and the roles of different stakeholders in a project. The courses are justified also because work experience following graduation for many of these students *will* include assuming leadership roles on software engineering projects. There remains the question of how to teach technical leadership to undergraduate students, many of whom have not had any related work experience. This paper addresses that question.

2. BACKGROUND

For the 2003-4 school year, the computer science (CS) department at Rose-Hulman Institute of Technology (RHIT) added a baccalaureate program in software engineering (SE). This new program included three courses introducing students to various aspects of software leadership. Those courses also promised to help solve a persistent problem for RHIT—Over 90% of the graduates of both bachelor's programs would become a part of software development organizations, and many would go on to become technical leaders and managers in the software field without necessarily gaining an advanced degree as additional preparation. More than 1/3 of the department's CS graduates over the past ten years have assumed leadership positions in their field, with titles such as Project Manager, Development Manager, Chief Software Engineer, Software Architect, Director of Software Engineering, and Senior Systems Analyst. Many more perform technical leadership activities as a part of jobs like Lead Developer and Software Design Engineer. Yet less than half of these bachelor's graduates acquire additional degrees.

While a few of the rest gain management training in other ways such as a certification in project management, generally there appears to be an unmet need for leadership training among these graduates. Additionally, the further education which they do get may not be specific to leadership in software development; for example, an MBA or a master's in computer science which focuses purely on technical subjects.

One could argue that anyone with a bachelor's degree in computer science or software engineering is a *likely* candidate for technical leadership in software development simply by virtue of having this much background. According to the U.S. Bureau of Labor Statistics, only about half of the people doing computer programming in the U.S. have a college degree in any field (Bureau of Labor Statistics, 2004-05).

The question of providing engineering leadership skills to undergraduates is one which generally has fallen "under the radar" in discussions among educators. No ABET criteria for management, software architecture, or leadership exist for either software engineering or computer science (ABET, 2004a; ABET, 2004b). Indeed, management is an ABET criterion only for architectural and petroleum engineering, while leadership is a criterion only for construction. This differs from the ABET criteria for advanced engineering programs, where leadership *is* included (ABET, 2004a). Interestingly, these omissions also make the criteria for undergraduate engineering programs different from criteria for architecture in the building trades; there the National Architecture Accreditation Board specifies management and leadership as integral to an undergraduate's development of design and technical skills (NAAB, 2002).

The IEEE/ACM curriculum guidelines for bachelors programs in computer engineering, computer science and software engineering do not emphasize technical leadership (JTFCEC, 2004; JTFCC, 2001; JTFCC, 2004). In these three related sets of guidelines, the word "leadership" appears only as a part of the objectives for software engineering. There, one sees a goal to understand and appreciate leadership, not to apply knowledge or show mastery of it. The computer engineering and software engineering guidelines do list management skills as a part of preparation for professional practice. Requirements management, software processes and practices are included in the guidelines for CS and SE curricula, playing a substantially larger role in the latter. With both curricula, however, software architecture appears to be considered a non-people-related task, a subset of design work. In most of the uses of "management" in these guidelines, it would be possible to conclude that "leadership" was not an expected part of that. For example, in the detailing of SEEK areas and knowledge units in JTFCC (2004), the management core areas of planning, organizing, and controlling are listed, while leadership is not.

The bachelors in software engineering is still an unusual degree in the US (Bagert and Chenoweth, 2005), and most schools do not have courses in their CS departments which specifically address the subject of technical leadership. More often one sees a one-term to one-year course in software engineering, with as little as a week on each of the topics of requirements, project management and architecture, and little focus on leadership. CS undergraduates are also offered a capstone project in many schools, which also is true at RHIT. The amount of leadership experience provided in these senior projects varies a lot;

often each team has a single leader over its entire duration; everyone else is a team member; and only the team leader has regular contact with the project's client.
To create the SE curriculum at RHIT, the IEEE/ACM guidelines were enhanced by leadership topics derived from the course authors' industrial experience. The courses from this new SE program which promised to help fill the need for leadership education included software requirements engineering, software project management and software architecture and design. The first two of these were made a required part of the CS curriculum as well as of the SE curriculum. Software architecture and design became a required part of the SE curriculum, and a course which many CS majors took as a free elective. It is useful to describe here how each of these courses contributes to the building of management or leadership skills:

Software requirements engineering is a course about interacting with customers, clients and users in order to elicit, understand and manage their needs related to a software project, and to communicate those needs to software developers. In many software organizations this is a vastly larger role than just writing down requirements as someone states them, or translating their form. Requirements engineers must gain management cooperation to study workers whose jobs they may redesign, and the engineers must extract information skillfully from subject matter experts who are themselves used to playing a leading role but not necessarily used to describing that role. The most critical part of the requirements job is conflict resolution among dissenting interests, and leadership skills are required to reduce project risk. The requirements analyst represents the project to its outside stakeholders, and represents those stakeholders to the project.

Software project management deals not only with estimating and scheduling technical tasks but also with organizing and leading software people toward successful development. A course in this area is direct preparation of undergraduates for their likely first promotion into management. Project management is somewhat different for software engineering than for most engineering disciplines, due to the fact that different styles of development are allowed which incur more risk (Chenoweth and Yoder, 2004). The spiral development model is a good example – in it, development is allowed to commence even when it is acknowledged that the client, users and customers are unable to provide clear and complete requirements.

Software architecture and design includes the usual topics from the world of software design, such as design qualities, methods, and patterns; the course goes beyond this in that software architecture means overall technical leadership. The software architect is responsible for the technical success of a whole software project – his or her high-level design must work. It must solve the client's problem. The architect also must sell the design, as a solution to a problem, to all its stakeholders, from client to developer and tester.

Each of these three courses thus promised to help fill the need of preparing RHIT undergraduates for a career involving leadership and management in software development.

3. EXECUTION AND RESULTS

These courses were taught for the first time in the 2003-4 school year, primarily as required courses for juniors, with some senior CS students and others taking them as electives. Software requirements was taught in the fall, then software project management, and finally software architecture and design, in RHIT's three-term school year. Practicing of leadership roles was distributed across these courses using a common project as a vehicle. Here are some highlights from these courses:

3.1. Software requirements engineering

In this fall course students had to elicit and write real requirements for the multi-course project, while interacting with a client who was external to the classroom. The project was to create a graduation planning system, an electronic form which followed the real rules for graduation at RHIT. The clients were professors from CS and other departments, who were in fact experienced advisers and knew the system requirements for their major areas. The students did this work as teams of four; in each class they had different clients, leading to varying results even with the same project. The teams built requirements documents, including detailed use cases and a prototype, which they then had to defend in front of the class and the client. Almost all the teams were successful in building and presenting a sophisticated set of requirements addressing their client's needs as they had heard them.

The instructors found while teaching this course that students who already had some related work experience had, in general, a much different attitude about the course. The level of this work background varied. Many students were involved in part-time jobs, had done extensive co-oping or had held summer internships. A few worked full-time in software development while attending school. Almost all of these work-experienced students had been a part of projects large enough that they themselves had played a narrow role, and they had experienced in person the difficulties large projects have with communication and leadership. They had seen first-hand how key requirements which were incorrect or missing had caused project delays and rework; they were extremely eager to hear how to do requirements engineering effectively. This was no surprise – most software projects fail in some substantial way.

In contrast, roughly half of the students in these classes had *never* done real software development. They had perhaps never been faced with systems so complex that engineering approaches had to be used beyond intuitive application of the codified CS body of knowledge. The software requirements course was the first course that was a part of their major but which was not clearly built upon first principles. The need to appeal to heuristics and engineering best practices felt "not right" to some of them. The instructors heard from them that the course had no place in a computer science major because it was empirical, not inherently analytical. It was a very rude shock to these students that things had to be done a certain way because someone called the client wanted it that way, and if the student thought of a better way, the client might simply exert their authority and say no.

This discrepancy in reactions was a problem which had been anticipated. In general, undergraduate education in computer science has focused on the deductive application of first principles; students have not often been faced with systems large enough that other engineering approaches become preferable. Nor have they been faced with situations where someone else's value system has an effect on the work they must turn in for a grade.

A remedy for this problem which RHIT had already begun to apply was a radical one impacting the overall CS curriculum. RHIT had started teaching the three core courses in its computer science program as if they also were software engineering projects. Even in the freshman course in object oriented programming, students had to write requirements and draw designs for review, work as teams, use the spiral development methodology and employ software development environments. In the second course in programming, students were being introduced to projects where they were each other's clients, not totally in charge of how they solved a general problem presented by the instructor. The heuristic for this effort was, "How much of software engineering can we push down to an early stage in the program?" Students in this first software requirements course, now mostly juniors, had not had as much of this new approach when they had taken these earlier courses.

In the software requirements course the instructors adopted several tactics to help overcome the lack-of-experience problem. They mixed students having development experience in with those who had none. They created the project to be something related to school and which solved a problem familiar to all students. They included at the end of the course an actual programming problem – the creation of a GUI prototype; for some students the fact that they got a chance to "sling code" brought home the reality of the rest of it. The two instructors who taught the course both had had real-world software development experience and brought to the class examples of requirements, war stories, and exercises which tied students' imaginations to the realm of software development. The instructors provided supplementary text materials with examples and case histories. As a sample course material, the first class homework was to write a problem statement for getting a job which the students actually might want upon graduation, extracting requirements from Monster.com position descriptions.

The instructors had hoped that the use of professors as clients would aid in resolving another anticipated issue. This was the issue of gaining as much time as possible for students to interact with their clients while they are engineering requirements. Immersion in a realistic work environment is the learning situation most likely to change the values of students not having outside experience. The professors' availability was more limited than would have been ideal. In the end the instructors tried for one meeting a week between each team and these clients and were lucky to get that. The ideal would have been more like having interactions with the client every day. Constant contact is a tactic used in very well run projects which are creating something novel to both the developers and the client, such as a new kind of product.

It is the standard preferred in the Extreme Programming method of software development (Chromatic, 2003; Beck, 2005).

3.2. Software project management

In the second course students built a model for a project plan from knowledge gathered about available people and tools for developing the multi-course project. Most of the teams were successful in creating a plan which dealt with risks and contingencies, and which emulated real-world project issues.

This course found the same variance in student attitude, based largely on their depth of software development background. Students who had worked on projects larger than school sized ones were much more able to perceive value in the techniques of project management. The instructor used similar teaching methods to relate the unfamiliar practices and heuristics to students without software development experience.

This class ran into an additional problem – Some students for whom this course was a requirement had actually worked as project managers. This required an accommodation in a new direction, for those already acting as technical leaders and wanting more advanced subjects. The instructors dealt with this reverse problem in two ways: The most experienced student was given special assignments in lieu of the regular class. These assignments included creation of a project plan with more detail and nuances, and a risk assessment activity for a senior project which had some hidden risks. For the other students who had had some experience, the instructors supplemented the regular project management class with weekly seminars where they discussed special topics of their own choice – issues related to technical leadership. A sample topic was, how to handle a double client situation, when only one of the clients is cooperating with the project.

3.3. Software architecture and design

In the third course students created an architectural framework and specification for the multi-term project, presenting their design to outside clients and to potential developers. As about half the material, this one-term course included design topics like the "Gang of Four" patterns (Gamma, *et al.*, 1995). The goal of also covering a substantial amount of software architecture was rather ambitious. Indeed, it is unusual for a first course in software architecture to include the actual building and testing of a system design.

The course was sequenced so as to stimulate leadership, by providing progressively greater autonomy to student teams. This agenda was built into a wide range of team design and architecture activities, including the term project. The early weeks of the course focused much more on learning by doing individual assignments and readings, by attending lectures and participating in discussion. It was necessary to get through most of the design theory and text materials very quickly. This left the rest of the class time for the students to create an architecture document and then start to implement it.

In this course the instructor tried a variety of accommodations to increase the learning of all the students. He slowed the initially planned pace of the course, adding material designed to enhance the understanding of architectural subjects by all students. Once again, being able to do something more analytical appealed to many students, so more time was given to exercises in implementing design. The instructor included interactions with other classes – for example, the students in this class acted as consultants in a design project for a sophomore level CS course. And he included a reverse role-playing situation – those same sophomores participated in evaluating the designs done by the software architecture class, acting as "developers" who had to understand how a team's design worked in order to implement it. This test of leadership was very interesting to observe, because it required underclassmen to evaluate the work of their superiors.

For this class, records were made of prior student work experience versus class performance, so that results could more carefully be analyzed. Perhaps because the course was a free elective for CS majors, required only for SE majors, a much higher percentage had had related work experience: 79%.

On both their team projects and their individual work, performance results of the students were clustered into a high achievement group and a more modest achievement group. Overall, the second group scored a full letter grade lower. Differences between these groups were exaggerated on individual assignments, while being milder but still present on team assignments. (Some teams represented a mixture of these achievement groups.) In the high achievement group, all but one student had had prior software development experience. Three of the four students with no prior work experience were in the more modest achievement group. Additional factors appeared to contribute to the performance seen in this class, such as whether or not students generally had high academic achievement.

4. CONCLUSIONS AND RECOMMENDATIONS

The motivation for introducing leadership education into the undergraduate CS curriculum was to close a gap, one created by advancement of RHIT's CS graduates into leadership positions without getting an advanced degree as preparation for that role. The gap-closing was accomplished by offering courses to these students taken from the new SE program; these courses did provide education in software leadership. This paper described some key outcomes and learnings from the first year of experience at this endeavor.

4.1. Effective role playing

In retrospect, the most difficult part of making the requirements course successful was maximizing the amount of time the students spent with other people playing the roles of customers, users or clients. Those interactions are where the students get to practice their leadership in a safe environment. For students lacking related work experience,

only immersion in the actual task compensates for this lack. The instructors had teams try to meet with their clients once a week if possible. This was not enough, though it did emulate the rather limited way in which some software projects gather requirements. Students need to become immersed in the experience of working with a client, representing a project to the person who is explaining their needs. RHIT is still searching for a solution to this particular problem. When the course is taught without sufficient client interaction, students will learn to make up requirements on their own, those requirements which they could not get from a valid source. That is a terrible lesson.

The lack of face-to-face requirements time is a problem which impacts every undergraduate engineering course where requirements-gathering is employed. Because the projects are not "for real," real clients do not want to spend the usual amount of time explaining and negotiating the requirements with an engineering group. The clients' organization will not be impacted, or they will not really have to sell the project's output as a product. As senior projects, for instance, industry usually offers non-critical activity to students, because of their lack of control, the higher risk it won't be completed, and other factors. This leads to the industrial client's not wanting to spend the amount of personal time getting the requirements right that they would spend on a mainline project for their company. In particular, the confrontations where leadership and negotiation skills are important tend to be avoided.

One possible solution to this deep problem is to utilize outside clients from nonprofit or volunteer organizations to interact as clients for these projects. This already is done in senior projects and in multi-term projects such as EPICS (Jamieson, *et al.*, 2001). Nonprofit clients tend to be less sophisticated and less demanding than industrial clients, yet they often are willing to give more time to students working on a school project. Other options for client representation in requirements-learning courses may include having class assistants or students from other CS or SE courses play the client roles. Alternatively, each student taking the class can play multiple roles, being a requirements engineer for one project and a client for another project. Such networks of class roles surely must be devised with some attention to clarity.

4.2. Mixtures of work experience

A major concern which emerged in all three courses was how to use teaching methods allowing for the wide range of related work experience which undergraduate students have had at the time they take these classes. Key accommodation methods which the instructors found useful included:

- Supplementing text materials with examples and case histories;
- Offering separate discussion and help sessions for novices and for the experienced;
- Organizing team activities so as to provide gains to both groups, such as by mixing experience levels on a given team;
- Slowing the pace of the course, substantially below what the work-experienced students might demand;

- Mixing-in material which appeals intuitively to the less experienced, such as programming exercises that make design theory more real-feeling to them; and
- Tying these subjects to things which are familiar, such as projects based on school experience.

In the software architecture class, records of work experience showed that, even with these efforts, students with prior work experience tended to be higher performers. Thus, students should be encouraged to plan for work experience voluntarily, before taking this class. The need to slow down the pace in this particular course resulted in a revision – During the next year software architecture and design will become two courses at RHIT. The material of architecture and design will still be combined, in each term of that course sequence.

At the same time, RHIT attempted to close the gap from another, more radical direction, by including increasing amounts of software development practice into beginning CS courses. This effort, it is believed, will help alleviate an underlying issue for students having no related work experience. The issue is that such students tend to value intuition, analytical thinking and first principles; they tend to devalue empiricism and the pragmatic methods which underlie real software engineering and its technical management. Appeals made by the instructors to these students while they are taking classes on technical leadership are important; but the feeling of a change in values makes these courses difficult for the work-inexperienced student. It helps for significant pieces of software engineering to be introduced earlier in the CS curriculum. That introduction can reorient students' personal value systems toward being open to best practices and heuristics.

The issue of work-inexperienced students in engineering classes seems especially difficult to resolve in regard to their comprehending how the social environment of the real work differs from the social environment they are immersed in at school. In introducing one of the standard texts on human-machine interaction, Gary Perlman says, "Many students will not have the motivating experience of seeing projects and products fail because of a lack of attention, understanding, and zeal for the user" (Preece, *et al.*, 2002). A part of the problem surely lies in preparing these students to understand a wider range of people, the problem we described as a need for effective role playing. Neither clients nor systems users often have the culture and values felt by college students. Beyond this difference, as Perlman suggests, real work activity inspires students in ways usually missing from classroom experience. Students are used to succeeding in their coursework. By the time they are juniors in college, successful engineering students at selective schools may rarely have failed at anything they have tried. Almost all of what they have dealt with could be brought under control simply by increased intellectual effort. College classes and curricula are planned to be intentionally progressive, and students have less tolerance for false starts and wasted effort. To students who haven't been through such lumpiness on an outside project, a course filled with reinterpretations and rewrites can be demoralizing.

In this last regard, it is possible that the lack-of-experience problem is inherently hard to fix via formal education. Students tend to expect and instructors tend to deliver courses which are full of this feeling of incremental progress, presumably on a path leading to great things. This obligatory class atmosphere is much different from that of real software development shops where, "Good judgment comes from experience and experience comes from bad judgment."[1]

REFERENCES

ABET. (2004a). *Criteria for Accrediting Engineering Programs: Effective for Evaluations During the 2004-2005 Accreditation Cycle*, Engineering Accreditation Commission, Baltimore.

ABET. (2004b). *Criteria for Accrediting Computing Programs: Effective for Evaluations During the 2004-2005 Accreditation Cycle*, Computing Accreditation Commission, Baltimore.

Bagert, D.J. and Chenoweth, S.V. (2005). "Future Growth of Software Engineering Baccalaureate Programs in the United States," To appear in *ASEE Annual Conference & Exposition*, p. 4.

Beck, K. (2005). *Extreme Progamming Explained: Embrace Change*, Second Edition, Addison-Wesley, Boston, pp. 61-63.

Bureau of Labor Statistics. (2004-05). *Occupational Outlook Handbook, 2004-05 Edition*, Computer Programmers. U.S. Department of Labor. On the Internet at http://www.bls.gov/oco/ocos110.htm (visited January 28, 2005).

Chenoweth, S. and Yoder, M.A. (2004). "Project Management: Electrical Engineering vs. Software Engineering," *ASEE Illinois-Indiana Regional Conference*, 2004, p. 1.

Chromatic. (2003). *Extreme Programming Pocket Guide*, O'Reilly & Associates, Inc., Sebastopol, CA, pp. 36-38.

Gamma, E. *et al.* (1995). *Design Patterns*, Addison-Wesley, Reading, MA, 1995.

Jamieson, L.H., *et al*. (2001). EPICS: Serving the Community Through Engineering Design Projects, in Learning to Serve: Promoting Civil Society Through Service Learning, L. A. K. Simon, et al, editors. Norwell, MA: Kluwer Academic Publishers.

JTFCEC. (2004). *Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering* (Final Curriculum Report). Joint Task Force on Computer Engineering Curricula. IEEE Computer Society and Association for Computing Machinery.

JTFCC. (2001). *Computing Curricula 2001: Computer Science* (Final Curriculum Report). Joint Task Force on Computing Curricula. IEEE Computer Society and Association for Computing Machinery.

JTFCC. (2004). *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (Final Curriculum Report). Joint Task Force on Computing Curricula. IEEE Computer Society and Association for Computing Machinery.

NAAB. (2002). *National Architecture Accreditation Board 1998 Guide to Student Performance Criteria* (with 2002 Addendum). National Architectural Accrediting Board, Washington, DC, p. 10.

[1] Attributed to various people, including Fred Brooks.

Preece, et al. (2002). *Interaction Design: Beyond Human-Computer Interaction.* John Wiley & Sons. Forward by Gary Perlman, p. xxi.