Multi-team Projects for Introducing Software Engineering

Steve Chenoweth (chenowet@rose-hulman.edu), Chandan Rupakheti (rupakhet@rose-hulman.edu), Sriram Mohan (mohan@rose-hulman.edu) and Shawn Bohner (bohner@rose-hulman.edu)

Rose-Hulman Institute of Technology

Abstract

Full-sized industry-related and service learning projects can be tackled by coordinated teams, to provide the enthusiasm of an authentic project while undergraduate computer science and software engineering students learn the principles of software engineering. Now in our second year of this experiment, we have found that having one large, realistic project for a whole class more closely resembles industry experience. Furthermore, the single focus of the class generates many examples and activities applicable to every student, as they learn topics like requirements, project management, architecture, and design. The project size lends itself to full-year or multi-year duration, adding to the realism of the student experience. Regular client interaction can be rolled into the classroom experience (such as via Skype and Google+ Hangout). We report on differences that must be considered in this curriculum approach. For example, project and client selection are crucial steps toward success of the class, and critical interactions occur among the coordinated teams.

Introduction and historical background

A key challenge for most engineering disciplines is to give students a "near real-world" experience while in school, where they can refine knowledge acquired in the classroom and develop skills necessary to succeed in the profession. For software this challenge is exacerbated by the fact that systems which need the engineering discipline are usually large enough that they require teams of engineers to work together to produce the software, and to integrate it with other teams. This is the aspect that we focus on in this paper – projects with coordinated teams. How can students identify their misconceptions early, about how to work in this complex environment, without dire consequences that they might face in their first real job? We would like for them to learn the human and logistics sides of this organizational complexity, in school, where they are "safe" and while coaching can be administered. Such effort makes the classroom experience even more beneficial, both to the students and to their future employers.

Computer science first, then software engineering

Building on computer science (CS) foundations like representing and processing data, software engineering (SE) concentrates on developing/evolving software over time, addressing scale and quality principles. SE is still a nascent discipline and, as such, has fewer canonical methods for developing software systems. The field is growing so fast that relevant technologies used to develop software have short half-lives and students must do a great deal of hands-on learning

and cooperative learning in teams. To undergrads, it lacks the verisimilitude of CS because of these shallower underpinnings.

Our pervasive endeavor to teach teamwork

From their first software class onward, our students at Rose-Hulman are exposed to sharing their effort with other students – initially in small group activities in the classroom, evolving to team homework and projects, and ultimately to the capstone senior design project in their senior year, collaborating with three or four other team members to develop a substantial project for a real-world client. We see achieving via teams as one of our key learning objectives, because in industry, software engineers seldom function for very long in isolation.

Over the years, we have striven to give our students project experience that more realistically maps to the world they will function in. Initially, faculty who had experience in industry devised projects, and all the student teams in the class would do the same project. While this made it easy for instructors to set up and assess learning objectives, without having reified clients the method did not reflect career projects very closely. To students, the projects felt fake.

We then shifted to projects with outside clients, which were of a size that, we guessed, a team of four students could complete in a year. Most teams did deliver a system, yet the projects were necessarily small and often were not required to integrate well into the client's own environment.

So, we moved to providing two major software project experiences. First is a yearlong junior project where students take a sequence of three courses (Software Requirements Engineering, Software Design, and Software Construction and Evolution). The sequence serves as a medium for reinforcing previous CS concepts and principles, while introducing the key SE skills. This prepares them for the second project – a more substantial and autonomous capstone senior design project in their fourth year.

The present study builds on that basic course structure. It is an advanced exercise in generating teamwork skills in undergraduate engineering students. The students are all juniors majoring in CS or SE. The ambitious goal is for them to tie SE skills, as they learn these formally in the classroom, *to large, multi-team projects that require SE skills to succeed.* All the projects are real, with clients who have more than a passing interest in their success. Integration with the client's real-world environment is a major component of the projects, adding both difficulty and face-validity. In one of this year's projects, the specific project goal is integration of multiple tools on multiple devices for the client.

There is beyond-the-usual risk in this large-project approach – more stress and strain, more temporal and architectural dependencies, and multiple constituencies. At the same time, a substantial project like this proves the value of the processes used, in ways that typical school projects do not. It requires that students not only work well on a team of four or five, but that they also synchronize their work with other teams in rather precise ways. The teams rely on one another to produce understandable and coordinated matching artifacts.

We are now half-way through our second year of trying that avenue for teaching SE. This paper is an experience report on that enterprise.

Should an engineering department teach this?

At our school, two of the three junior year courses, where students work on these projects, are required of CS majors as well as SE majors. This is in addition to both majors' following-up the junior project experience with a full-year capstone senior project. Furthermore, all but a few of our CS courses culminate in a multi-week group project. That includes courses in the beginning CS sequence. We have a rather larger-than-normal departmental dedication to teaching teamwork. We believe we are in synch with *The Engineer of 2020* report, that, "the engineering profession recognizes that engineers need to work in teams."[2] Our unusually high value for delivering this "soft" outcome deserves an explanation.

Purely technical advances from research are allowed to move into undergraduate engineering education in an orderly way; say, a new, faster, algorithm discovered in mathematics or CS. The need for teamwork is a bit different. When the stimulus comes from outside research like this, from outside academia, in fact, then an automatic curriculum adjustment is not guaranteed. We believe this adjustment-source issue has been a factor making it difficult to get the proper amount and type of teamwork learning into baccalaureate engineering programs.

From a systems perspective, undergraduate education institutions get high school graduates and produce able learners capable of greater personal and career contributions to society. In science and engineering schools, this system-like transition is even more official, given that our graduates are targeted to a discipline, often to help shape a specific industry. It has face-validity that, on one end, our undergraduate pipeline *should* look familiar to incoming high school graduates (i.e., they have prepared for entrance into college), and on the other end the pipeline *should* make their chosen industry look familiar when they arrive there from college. We strive to impedance-match our inputs and outputs. The matching effort works in a lopsided way, however. We ourselves can see the implications of mismatching the expectations and capabilities of incoming freshmen (i.e., lack of prerequisite knowledge or maturity). On the other end of the pipeline, the graduates more-or-less disappear from our campus. How can educators better condition our graduates for the workforce and the expectations of an industry culture?

Providing rich experiences in teamwork is an example of an output that would closely meet the needs of today's engineering. Gone from the industrial landscape is the engineer who needs few social skills as he works alone at a drafting table. Almost all engineering is accomplished now via interactive and interdisciplinary teamwork. On large projects, the teams are complex; the interactions between them often represent the interfaces between the subsystems each team generates.

Academia tends to respond slowly to changes like this; perhaps due to our separation of concerns trying to keep a principled mindset. Our engineering curricula, especially, are always "full." This is commonly cited as reason-enough not to do anything additional, particularly when the graduates are already being hired. If suggested new material falls outside the traditional scope of our departments, it can be delayed or rejected out-of-hand, as well. Regarding teamwork in particular, we also may be lulled into not striving to produce this in students because academia itself relies less on seasoned teamwork than engineering industries now do. Teamwork further is

a slippery place for student evaluation, where educators must manage potential cheating; hence, team assignments have some negative baggage that may produce barriers.

Even with well-meaning department advisory boards (members from industry and other educational institutes), subtle changes like this may go undetected as board members (often alumni) harken back to the good-ole days and assume teamwork will be learned in companies or on internships.

Increasing the influence of industry, beyond this historical advisory board level, may be resisted by a department. We tend to see ourselves as originators of ideas, prime movers of both education and of change in the industries we serve; change goes from our world out into that other world. Thus we see reported statistics like University of Michigan's, "that 72% of recent undergraduate alumni rated 'ability to function on a team' as extremely important in their professional experience, but only 47% felt that U-M provided excellent preparation in this regard."[3]

One may need to stir the mix of underlying rationales, to cause more to happen in a department. Having an effective diversity of perspective requires true diversity in backgrounds. Sprinkling a sufficient number of faculty from industry, into a department, does improve this impedance match for graduating students. Like some advisory board members, such faculty have seen new hires fail before their eyes, and as faculty, they now have the direct means to do something about that. Teamwork is an example of a skill that makes all the difference to new hires, and it is one which is resisted as being "a soft skill" by college engineering departments, unless enough of them have seen the results of bad teamwork in industry, for themselves.

The complexity of a "soft" subject

Of course, it takes more than desire to teach teamwork effectively. In these areas, "Our teamwork and leadership vocabulary is generally underdeveloped and our teaching strategies are also behind what leadership and team researchers currently know."[4] Use of resources in the other direction also has been done. Goldfinch, *et al* showed that group working skills in undergraduates could be improved by bringing in the final arbiters, their future employers, to coach the students.[5]

Teamwork is not a simple topic, whether "soft" or not. And teaching it, in the artificial classroom world, has even more wrinkles. Teamwork is, "usually challenging for inexperienced students who are struggling with technical as well as program management and team building issues. There also appears to be a general lack of ability by students to function on teams." As Massie and Massie described it, students cannot be expected to learn this on their own.[6]

It requires skill just to understand how teams are doing. Teamwork effectiveness can be self-assessed by team members, by applying standardized surveys, by measuring factors associated with high-functioning teams, or by valuing the team's results.[7]

At Rose-Hulman, our Computer Science and Software Engineering Department has been working to better teach teamwork for over 25 years now. Our first "serious push" to get teamwork into everything was in 1990. The large, multi-team projects we describe in this paper are the latest wrinkle in that long-term endeavor.

Like many schools, Rose-Hulman had experienced problems in getting SE topics across to our software students. Processes and practices were a hard sell to them partly because the work students had done up to then, and the projects they were given to accomplish during SE classes themselves, did not really rely on these processes and practices, in order to be victorious. Often, one night of heroic programming and testing by a single motivated team member was all it took. This haphazard approach has intrinsic rewards for the students, and it is bad practice for the orders-of-magnitude-harder projects of industry. We were making our code-slinger students eat spinach, on the promise that the "vitamins" would make them grow up healthy.

Methods – Moving to the multi-team approach

The large-project solution to this conundrum, of proving the value of SE principles to undergraduates, was developed over the summer before our 2012-13 school year. We had been using outside clients in our junior sequence of SE classes, for five years, as a means of adding realistic engagements to the student experience. One client that we were talking to, during that summer, had in mind a much larger project than a single team of four to five students could accomplish, even over the whole school year. It was a great project, creating a very practical software tool for him and other people engaged in similar work. The project involved building of a server platform and of applications running on multiple devices. So an additional incentive, for us to do a larger project, was the desirability of this particular activity. The project, with its one client, ran during the full 2012-13 school year, and the system was delivered at close to completion, at the end of the spring term. During the first two terms of the year, the project represented a single, common activity for one whole section of a course. There were 20, 17, and 10 students working on the project simultaneously, over the three progressive terms of the school year. The lower number of students in the third term was due to the fact that this course is required only for SE majors, not CS majors.

Over the summer of 2013, we recruited four clients to do similar, large-scale projects. During the fall and winter terms, these have represented the sole project for each of four sections of the SE courses taken. For the fall term, the four sections had 21, 23, 21, and 22 students; for the winter term they have 20, 15, 25, and 25, respectively. We have yet to organize them for the spring term this year.

Operational considerations

In both school years, the clients have met regularly with the entire class. This has averaged one meeting a week, either face-to-face, or over Google+ Hangout, for close to 50 minutes each time. Clients have discussed plans, reviewed artifacts and parts of the systems developed by their classes, and provided feedback on those.

All the projects have been managed via a Scrum-style process.[1] This process was adopted with the goal of showing the real clients early developments, and gaining their feedback. We believed it also enabled the students to do coding and testing early on, and to see the value of setting up processes that complemented rapid development and repeated deliveries.

Both years, the nominal course content was Software Requirements Engineering in the fall, Software Design in the winter, and Software Construction and Evolution in the spring. Had we

used a Waterfall process, we could have taught these subjects in a just-in-time fashion for the project development work. However, we felt that the advantages for using an agile process outweighed this more orderly approach.

In previous years, we had not so seriously pushed, to have each team's project be fully agile, simply because this meant they had to do things like development early-on in the project, before we would have taught all the topics they would need to make use of. Agile is not really for beginners; we believe that anyone teaching SE subjects runs into this same problem: If the team is supposed to deliver something with real code in it, say, a month into the project, then how do they do that using the parts of SE which have not yet been taught? For the much larger projects we were now going to be using in class, we felt it was more important to do the early, agile deliveries. Thus, there would be some parts that needed to be reworked, once students discovered better ways to do things. This would be especially true of code done while still studying requirements, code which would not take advantage of things like design patterns and solid architectural principles learnt later.

Students are responsible for the client meetings. In sections where the clients are remote, we have a given team run the Google+ Hangout meeting on behalf of all sections. In all cases, each team of about 4 students is responsible for presenting and describing progress to the client, for their part of the work. They also are responsible for capturing client reactions, and for documenting agreed-upon next actions.

For special coordination needs, each of the small teams has a person responsible to meet or talk with all the other teams. This has been done, for example, to assemble all the requirements into a single document, for deciding an integration process strategy, and for making decisions about common back-end interfaces. Similarly, any team discovering valuable information from an outside source has a responsibility to pass that on to everyone else who needs to know. So, someone from the team with news needs to do that coordination.

Reducing uncertainty

We also attempted to anticipate issues that we thought especially added to the risk for the approach we were taking. Where possible, we had multiple clients for one project, or a client who could step in if anything happened to the primary client. We were clear with the clients that this project would take a couple hours a week of their time, on average. We had extensive discussions with them about the scope of the project. Suppose that what they wanted turned out to be much less work for our students than it was originally projected? This is worse from a pedagogical perspective, than if it turned out to be more work. Or, suppose that the project was ideal for one of our three courses over the school year, but terrible for another? That could happen. For instance, the second course is all about object-oriented design. It was possible that the project would end up being best done with some non-Object-Oriented type of tool, such as JavaScript. With smaller projects, such an event would be a tragedy only for four students, not for a whole class.

We had to set up the methods the students would use for running their projects. We could not have them employing different tools for saving artifacts or managing code, say. Teams had to see what each of the other teams were doing, which might interfere with ideas of grading them

without knowing how others did something. Interactions with other teams had to be an expected part of their process. It was tempting to make common project success or failure the main basis for the course grade for everyone! Further, with so many interdependent projects (a project of projects), it was tempting to become the program manager for the integrated projects – potentially encouraging the students to take less initiative and wait for direction from the instructor!

Team processes like code reviews and testing practices needed to be in place from an early time in the first course. The database had to work for all the teams writing apps against it! Software industries use a variety of project management tools to facilitate these activities in their own software development processes. Due to the scale of our projects, we soon realized that our projects were no exceptions. We adopted several tools to facilitate collaboration among teams and their client. We use Google Groups and Google Docs to record meeting notes and for other project-related discussions. We use the Github Wiki to document the design of the project and other project artifacts. Github is also used for source code management and bug tracking. Trello boards are used for sprint planning and resource allocation. Use of such tools thus gave our students a feel of the software development process practiced in the real world!

We needed to pay attention to the usual issues of composing and running teams. How are known skills allocated across teams, for instance? And how are teams rewarded, with grades in our case, when they all depended on each other in ways that are excruciating to separate?

Results

The opening result was that students in all the sections were excited about the idea of doing a substantial project which their client cared a lot about. The apparent advantage was that the client would stay engaged throughout the project. In some of the sections, we additionally managed to make the project something that students themselves cared about – it related to their academic life at Rose-Hulman! The size, importance, understandability, commonality and relevance all played a part in this enthusiasm.

Roles of the instructor

For the instructor, these projects were more difficult to manage than small projects. There were more pieces to worry about. And, the instructor ended up, in each case, being roughly the class project's resource manager – the first-level manager who handles personnel as well as project matters. For example, with our previous, single-team projects (four or five students on a single project), the students were responsible for arranging and holding all their client meetings, reporting the outcomes to the instructor. With all teams in the class involved, the instructor had to take over making sure this happened – and the only logical time for the client meeting was during one of the weekly classes. Thus, the client had to be available at such a time!

In this sense, we lost some of the independent responsibility that had been forced on smaller teams, to do all their coordination work with the client.

If a client could not make a particular meeting date, this impacted everyone, typically for a week. In several cases where this happened, we were lucky to have a second client who was fully informed, and who took their place at the meeting with the class.

The instructor also now played project manager, in terms of handing out work on the project to the teams. When we had had simple teams of four or five, typically one student played the project manager or scrum master role, deciding who is supposed to do what, next. Now, on some of our large projects, there was an obvious way to divide the work over the whole class, and on others there was not; and this division fell upon the instructor to do, regardless. The first year, our project had different devices upon which the same application was to run. Three teams were given the task of doing this application on a particular device, while the remaining team got the task of creating the server-side software. The second year, for two of the four large projects, there was only one interface, so the application itself had to be divided among the teams.

On some of the projects, it was possible for the instructor to restructure which teams were doing what, as is done regularly by project managers. This was doable because all the teams were familiar with the whole project. Suppose there was a irreconcilable difference between two team members, or there was a team member who dropped the course; the instructor could reassign team members among teams, to best address the situation.

Complex coordination

At the end of our fall requirements engineering course, we needed to integrate all the documentation for each large project. This was accomplished by appointees from each team, working together as a cross-team. The clients were not happy with the results; mostly, some teams' representatives worked harder on the integration task than others. This left those contributing the most with the option of finishing others' work themselves, or knowing that the final result would be uneven. As instructors, we elected to allow the clients to see an uneven document, rewarding or punishing those who were on this integration team for their level of contributions. Further, the students would have an opportunity to reflect their concerns in peer evaluations, which served as a governor on student participation in the project teams.

Student outcomes

Student evaluation comments at the end of the fall course showed generally that students appreciated having a larger project. Probably the most important aspect of this was that they maintained their excitement about the project over a long period of time. And that was surely fueled by having the same project shared by a whole section of the course. The classes met four times a week, and every day's class felt like a mass project get-together – Better even than the daily standup meeting which is standard for Scrum.

We allocated grades by using several stratagems to deduce individual accomplishments: (1) We also gave individual work to do (homework assignments and exams), to verify that every student could accomplish each of the SE skills. (2) We used records such as weekly status reports and GitHub commits to track the activity of individuals on each team, and to provide the launching point for individual teamwork counseling. And, (3) We got feedback multiple times from team members, in the form of peer evaluations, using standardized questionnaires.

All students did individual journals, and they turned these in with each project milestone (or sprint). These were reflective documents demonstrating critical thinking, which they were taught to do like engineering journals, not like blogs.

We further tried to apply a "fairness doctrine" to the project success. As in these projects, students will encounter in their careers that sometimes it is not the fault of the development organization when a project fails. Certain of our projects are inherently speculative – the first of their kind – and clients change their minds enormously about what they want, in addition to things turning out not to work, technically. And large projects probably will not be completed entirely to a client's satisfaction, in synch with the end of the school year.

Discussion – Future developments and studies

We are now in the middle of our second year of running these large, multi-team junior projects, while teaching fundamentals of SE process. We believe that the idea has sufficient promise to continue at it, exploring refinements and variations, for a good while longer.

Agile is fragile

The issue of how to introduce topics to people unfamiliar with them, out of sequence with performing these tasks on a project, remains an issue for using agile development methods in a project accompanying SE principles. This problem is shared, also, for smaller projects. In our own history, we simply ignored it when the problem was an aspect of projects owned by smaller teams. We either let a team move ahead, to deliver early, on their own, or else nudged them toward more of a waterfall approach.

A new role model

As instructors, our main contribution to the class may be demonstrating what an effective resource manager does, so that, acting through the class-sized full team, we lead them to achieve difficult technical goals. Being able to do this role well probably requires some background on the instructor's part. The traditional presentation of us as classroom teachers is receding in comparison, though it has to remain because of the course outcomes. A possible long-term goal is for it all to be problem-based learning (PBL),[10] with most skills developed by students on a need-to-know basis. That PBL likely will be achieved, in the future, more via self-study than lecture.

Tricky grading will persist

One could speculate as to how well the idea of grading school assignments goes with providing the large-project experience to students. Neatly defined individual work no longer exists, on the project itself. The latest insult to clean grading is the fact that each team's work is often needed to be shared before grading essentially by the whole class, as a project resource, as soon as it is created. Also, overall project needs do not always get allocated evenly to individual teams – some have more to do. Can a whole class be failed, if the project is not acceptable to an outside client?

The future for SE education

Ultimately, the larger, integrated project-of-projects concept offers great promise for the learning outcomes in SE program. We are particularly interested in the transition from a "toy" project with four team members (an inadequate medium for teaching key scale and team aspects of SE)

to a more real-world, multiple-team integrated project that serves to highlight the integration, large-scale design coordination, and especially the team interdependency issues that arise in real software and other engineering projects. As we work through the challenges of managing the parts and we gain experience in how to capitalize on coaching students about the concepts we teach in the classroom, we are hopeful that the approach will be useful for the SE education community, and perhaps for the engineering education community as a whole.

All software projects tend to run over their allotted time anyway, and larger projects are worse. With such foresight, these projects can be targeted as multi-year to begin with, which further gives a realistic software maintenance side to what students must do in the following years. Service learning projects naturally fall into this multi-year category, as exemplified by projects done in Purdue's EPICS program.[8] Here, local clients tend to be relatively stable and accessible, their needs for automation are large and long-term, and these needs in fact grow over time as clients become aware of what can be done for their non-profit ventures via software. At Purdue, this "mission creep" is handled by having an EPICS team of 15 or so students assigned to a client, rather than to a specific project.

With very large projects, a junior year engineering experience like we have described could simply continue into successive years, with each year's students doing another "release." Continuity could be established by having the prior year's students, now seniors, serve as advisers, in addition to the obvious reliance on documentation and on instructor and client persistence. It also is possible for projects, planned to be two years in duration, to move with the students, from this junior year experience into their capstone senior design project. Such a plan would further allow for an opportunity, that some or all of the junior students could intern, over the summer in-between, at the location where their creation is being used, then return all-the-wiser to make it work for real!

These teamwork ideas do add risk to the certainty of delivering specific technical content in a course. The topics they learn most in-depth will rely on the project. Such is the nature of PBL. And, with a social goal like teamwork being elevated, something else will have less emphasis. Yet, spending time on teamwork does reflect the realities of the careers we are sending our engineering students into. For example, 900,000 of the 1.3 million software engineers and computer programmers in the US spend their time maintaining existing systems, not building new ones.[9] Maintenance involves a preponderance of testing time, and system testing is inherently a large-team-based activity.

References

1. Takeuchi, Hirotaka; Nonaka, Ikujiro. "The New Product Development Game" (PDF). *Harvard Business Review*, Jan 1, 1986. http://hbr.org/product/new-new-product-development-game/an/86116-PDF-ENG.
2. *The Engineer of 2020: Visions of Engineering in the New Century*, p. 43. National Academy of Engineering, 2004. http://www.nap.edu/catalog.php?record_id=10999.
3. Finelli, C.J., et al. "Student teams in the engineering classroom and beyond: Setting up students for success." *CRLT Occasional Papers,* Center for Research on Learning and Teaching, University of Michigan, No. 29 (2011), p. 1. http://www.crlt.umich.edu/sites/default/files/resource_files/CRLT_no29.pdf.

4. Mansour-Cole, D. “Updating the Leadership and Team Ideas We Present To Students.” ASEE IL/IN Section Conference, 2013. http://ilin.asee.org/2013/index_files/mansour.pdf.
5. Goldfinch, J., et al, “Improving groupworking skills in undergraduates through employer involvement.” *Assessment & Evaluation in Higher Education*, Vol. 2, No. 1, 1999.
6. Massie, D.D., and Massie, C.A. “Framework for organization and control of capstone design/build projects.” *Journal of STEM Education*, Vo. 7, Iss 3, Jul-Dec 2006, p 36.
7. Huyck, M., et al. “Assessing factors contributing to undergraduate multidisciplinary project team effectiveness.” *ASEE National Conference*, 2007.
8. See for example https://engineering.purdue.edu/EPICS/About.
9. Donald J. Reifer. *Software Maintenance: Success Recipes.* CRC Press, 2012. ISBN 978-1-4398-5166-1. p 1. These are 2008 statistics.
10. See, for example, the discussions at the site for McMaster University, a pioneer in PBL, at http://cll.mcmaster.ca/resources/pbl.html.