

Spanning the Gap Between Software Engineering Instructor and Student

Andrew J. Budd, Heidi J. C. Ellis
Trinity College, andrew.budd@trincoll.edu, heidi.ellis@trincoll.edu

Abstract – Many computing degrees have a project-based software engineering course where teams of students complete a project under the guidance of the instructor and possibly one or more teaching assistants (TAs). However many instructors are unsure as to how well course objectives for these project-based courses are being met and more importantly, how best to structure the experience to optimize student learning. This paper presents the contrasting views of the instructor and a TA on the learning that occurred in an undergraduate software engineering project course. The undergraduate TA for the course is uniquely qualified to support the course, having had several years of real-world software development experience. This experience allows him to straddle the gap between the typical undergraduate student perspective and the requirements of real-world software development.

Index Terms – Software engineering education, Student perspective, Project course, Real-world learning.

INTRODUCTION

Many computing degrees have one or more software engineering courses to convey to students the principles of software development. A number of these courses are project-based where teams of students complete a project under the guidance of the instructor and possibly one or more teaching assistants (TAs) [1-4]. Typical student projects range from applications with real clients [5] to modifying open source software [6] to service-learning experiences [7].

The motivations for teaching a project-based software engineering course include allowing students to apply theoretical knowledge, providing students with real-world experience, teaching team interaction skills, exposing students to all phases of the software lifecycle, to name just a few. These are laudable goals, however many instructors are unsure as to how well these objectives are being met and more importantly, how best to structure the experience to optimize student learning [8, 9].

This paper offers insight into the learning provided in one such project-based course by contrasting the instructor's views and goals of the course with those of the undergraduate TA for the course. In this case, the TA is quite unique in that while in many ways he is a typical undergraduate senior (including age and schooling), the TA has extensive background in software development, being

the founder of a small software company. The paper provides a brief overview of the software engineering course including approach, the project and deliverables. The paper outlines the instructor's objectives and observations as to how well the objectives were met. The meat of the paper presents the viewpoint of the TA including his ideas of what kind of learning occurred and how learning could be improved. The paper also identifies the similarities and differences in the instructor and TA viewpoints and discusses what we have learned from the approach as well as improvements that we will make in future offerings of the course.

THE ENVIRONMENT

Trinity College is a small, competitive liberal arts college located in Hartford, CT with approximately 2100 undergraduate students. The Computer Science program is small, graduating between five and fifteen students per year. This small class size affords a high degree of student-teacher interaction while allowing instruction to be tailored to fit class characteristics. Students may choose between a BSCS or a BACS both of which require thirteen courses (CS1, CS2/Data Structures, Discrete Math, Calc I & II, Systems Software, Foundations of Programming, Theory of Computation, Senior Project/Thesis, and four electives).

I. CPSC-240 Principles of Software Engineering

CPSC-240 Principles of Software Engineering is an upper-level elective in the computer science program. The prerequisites include CS1, CS2/Data Structures and Discrete Math. Given the fact that there is only a single software engineering course in the CS curriculum, the overall objective of the course is relatively general: to provide students with a broad understanding of the discipline of software engineering and its application to the development and management of large software systems.

The course meets twice a week (75 minutes each) for 15 weeks and class time is spent in discussion, completing exercises and performing software engineering tasks such as reviews. The texts used are Pfleeger and Atlee's *Software Engineering Theory and Practice* [10] and McConnell's *Professional Software Development: Shorter Schedules, Better Projects, Superior Products, Enhanced Careers*, [11]. The grading consists of three quizzes (20% of total grade), a series of 8-10 homeworks (20% of total grade) and a team project (60%). The project consists of nine different

deliverables ranging from an initial launch report to a final presentation.

During the fall 2007 term, eight students took CPSC-240 and the course was supported by an undergraduate TA. The eight students were divided into two teams of four to work on a project and we attempted to distribute skills and abilities equally across the two teams.

The project for the semester was an existing web-based Timelog project that allows students to log time in various activity categories. The Timelog project was originally developed to try to determine how students were spending their time in various courses, but the Timelog system could be used more generally to allow users to log time under various activities. The project was developed using PHP and MySQL and contained approximately 100 files including the Pear PHP library, 19 PHP library files developed specifically for the project, 13 PHP functional files, 25 HTML files, and additional JavaScript and CSS files. The Timelog system was developed using the Trac integrated content and project management system (trac.edgewall.org/). The Trac system contains a wiki where project information was stored, a scheduling component which allows milestones to be defined and scheduled on a timeline, and a ticketing system to track effort where tickets may be opened for specific tasks, assigned to team members and the completion of the tasks tracked. In addition, Trac is integrated with Subversion and all source code was managed and versioned from within Trac.

The TA for the course was one of the original developers of the Timelog project and therefore has a detailed understanding of the project. The two student teams were assigned two different sets of modifications to be made to the Timelog project.

II. TA Background

The TA for the fall 2007 offering of CPSC-240 Principles of Software Engineering, Andrew Budd, has a distinctive background. Andrew, now a senior at Trinity College, has been coding since the beginning of high school. Andrew is also a founding member of a small company that provides technological frameworks designed to support the deployment of large scale web based and web accessible applications in fields ranging from gaming to centralized social networking systems. Andrew has been active in both actual software development as well as the management aspects of the company. Andrew's knowledge of software engineering techniques and process, understanding of team dynamics, and management experience including scheduling and planning make him an ideal TA for the software engineering course. In addition, he brings the knowledge of the Timelog project which allows him to easily support learning as students modify the project.

Andrew is in the unique position of being able to clearly view both the student and instructor sides of the teaching relationship. As a student, Andrew is subject to the pressures of the typical undergraduate student and understands the context and constraints of a student trying to learn a vast

amount of knowledge in a short period of time. Andrew's software development background provides him with a real-world understanding of the knowledge that a student completing a project-based software course needs to absorb by the end of the course. Experience has shown Andrew the importance of the software engineering knowledge and skills that can be provided in the classroom. This combination of experience and computing background allows us to provide insights into how to better support learning for undergraduates in a project-based software engineering course.

Given Andrew's unique background, Andrew took on a more active role than is typical for a TA. Andrew became the Team Manager for both teams, providing an interface between the instructor and the teams. As Team Manager, Andrew met with the students once a week for 60 minutes for approximately 11 of the 15 weeks. While the instructor set the high-level deadlines for deliverables, Andrew oversaw the lower-level planning to accomplish these goals and served as a resource for answering questions and guidance during development.

In his role as Team Manager, Andrew's relations with students went beyond the typical TA's. Andrew acted as a facilitator to help students identify how to merge the disparate skills of team members. He also acted as a coach and an encourager and helped teams to organize their work so that goals were met in a timely fashion. Therefore, in addition to acting as a knowledge resource, Andrew had a direct effect on the operation of the teams.

INSTRUCTOR'S VIEWPOINT

In this section, the instructor provides her opinion of the course, its goals and how well the goals were achieved:

As instructor for the course, I want to convey as much of the essence of real-world software engineering to students as possible. I want students to take away some understanding of how to develop software for the real world. Specifically, I want students to gain:

- An understanding of the breadth and impact of the discipline of software engineering,
- A general understanding of software process including software lifecycle models such as the waterfall, spiral, and evolutionary models,
- An exposure to the role of project management including planning, control, organization, risk management, etc.
- an understanding of software requirements including requirements analysis, definition, specification and review,
- An exposure to software architectural styles,
- An understanding of the fundamentals of software design using UML as a notation including design evaluation and validation,
- An understanding of implementation issues including cohesion and coupling, modularity, coding standards, etc.

- An understanding of approaches to verification and validation including reviews, static analysis, and various testing approaches,
- An understanding of the issues involved in maintaining a large software system, and
- An understanding of the role of documentation.

The attainment of the goals above can be measured by the successful completion of the assigned project modifications by the two teams. While homework assignments required students to obtain a theoretical understanding of topics such as software lifecycle, architectures, requirements and more, working on the project brought home that understanding in a real and practical way. The use of the Trac system highlighted the role of and need for software process and project management. Throughout the semester, students performed various project activities in class such as requirements reviews, design, and constructing a test specification. These in-class activities provided the base of understanding for students to complete the project deliverables. The grades for the project deliverables for both teams were generally in the high B's, indicating that students were gaining an understanding of software development. In addition, seven out of the eight students rated the course as moderately challenging and one student rated it very challenging in the end-of-semester evaluations.

TA'S VIEWPOINT

In this section, the TA provides his view on the course and student learning in the course:

From the perspective of someone who has organized several commercial development teams, the content of a textbook on software engineering is far from sufficient to prepare a developer to perform adequately as a member of a real world development team.

Software engineering in the real world straddles the awkward positions of its highly 'post mortem' driven analysis of past projects, and the applicability of its development models and practices to an active team. In teaching undergraduate software engineering classes, it is my perspective that students should be placed into conditions which are as close to real world conditions as possible, so that the lessons offered during lecture sessions provide the maximum possible immediate benefit to the team. By doing this, not only are teams enabled to perform more efficiently, but students are also enabled to learn techniques in a hands-on setting that really brings the lessons into practice.

It is my perspective that the average undergraduate does not have the technical nor personal skills required to contribute to software engineering teams that focus on the development of large or existing software systems. Laboratory based introductory programming classes, particularly ones that provide large out-of-class assignments in addition to labs, encourage a mentality of 'hero programming' even more extreme than what is encountered in the real world.

In a typical Trinity College undergraduate CS1/Data Structures course, students perform weekly lab assignments with a random partner. Ideally, asymmetric levels of knowledge between pair members should be slightly rectified by this practice due to a sharing of information between the more experienced and the less experienced members of the team. However, as lab periods are at least two and a half hours long, and the students are dismissed as soon as they submit their assignments, there is a strong incentive for the more experienced member of the team to simply implement the solution without actually relying on the weaker member to contribute. This inequity is further exacerbated by the fact that labs are often written to accommodate the constraints of the weakest group of students. Inevitably, several teams in a lab will be comprised of two weak students, who will require at least the entirety of the period to implement a solution to a fairly standard problem. This precludes the possibility of assigning labs that the more experienced programmers will find challenging enough to require cooperation with their partner.

In upper level seminars, group work incentives evaporate as students are assigned difficult problems which they are expected to solve individually, frequently using new languages with which they are unfamiliar. Students who have chosen to continue in the major, typically the more proficient ones who have already been galvanized by the experience of rushing through labs while ignoring their partners, are then forced into a 'trial by fire' through which they must adopt a new language under very difficult circumstances. This process of trial and error learning under pressure helps to reinforce a tendency to want to work alone.

There are several reasons why this trial and error approach to learning might encourage students to want to work alone, not least of which is poor code quality. Students who are unfamiliar with languages such as C, but are still forced to implement assignments using said languages under stressful circumstances are usually not able to use the languages correctly. It has been my anecdotal experience that students can actually survive classes in Systems, Graphics, and High Performance computing without a solid understanding of how to reference and dereference `structs`, or allocate contiguous two dimensional arrays in C. Poor code quality leads inevitably to bugs that the developer cannot solve easily, and code which is so twisted that helpful classmates cannot even decipher its purpose let alone help to debug it.

The result of these early software development experiences is that, when faced with the challenge of working in a team to implement a given problem for a software engineering class, there is a strong incentive to assign all of the programming and design tasks to one team member while all of the documentation and specification "filling out" tasks are assigned to the other team members. This division of labor completely defeats the goals of learning good software engineering practice, and in some

ways may very well again reinforce the habits which we really hope to break.

Since I lead commercial development teams with some regularity, I have a keen interest in finding ways to improve methods for teaching software engineering in practical ways to teams of developers in order to make them better able to handle the challenges of real world team based development. Ideally, I would like to see far less “hero programming” as well as “hero documenting.” Hero programming results in a team that cannot maintain the software without the person who wrote most of the code. It also leads to massive inefficiencies as the productivity of the entire team is limited by the resources and abilities of the person who insisted on doing all the work. “Hero documenting” is as much of a problem as any future teams required to modify or maintain the system will inevitably end up with bad documentation. Forcing teams to write documentation for their code and modules is certainly essential, but ultimately useless if the documentation does not accurately reflect the implementation, as is often the case if documentation is created with little or poor communication among team members.

I. Team and TA Interactions

As the teaching assistant for the Fall 2007 offering of CPSC-240 Principles of Software Engineering, I worked closely with Professor Ellis to implement this real-world experience perspective into the class. I pushed to split the class into two teams of developers which would each seek to extend an existing system (the Timelog system) in different ways. Documentation for the Timelog system was incomplete, but the code was of a relatively high quality, and the team which had performed the previous principal coding was still available to answer questions. Many of the students in the class were not familiar with the underlying technologies used in the system such as PHP and MySQL, but I pushed for the teams to be split according primarily to ability and exposure such that each team had at least one member who had previously been exposed to at least one of the underlying technologies.

It would have been ideal to form two teams, each of which had to make a significant extension to the system. However, there was a serious gap in terms of proficiency at programming and software development experience amongst the members of the class, making the formation of parallel teams difficult. As a compromise, we formed one “A” team, and one “B” team. The A team focused primarily on rewriting a significant component of the system, while maintaining compatibility with all outside modules and libraries. The “B” team, in parallel, tackled a sizable list of needed improvements to the system, some of which were consequences of design deficiencies in its first development cycle.

I took the role of Team Manager, acting as the intermediary between the customer/manager and the two teams working under me. I did not once touch any of the code involved in the project, but would pass on requirements

and specifications in the form of highly detailed briefings, and was available to answer any clarification questions necessary to aid in the implementation of the system.

Each team was required to meet with me weekly for at least an hour to go over code, requirements, and progress. Teams were strongly encouraged to stay on target, and regular code reviews ensured that everyone was up to speed with what was going on with their project. Members of the teams quickly got the idea that they would sink or swim together, so after only a few sessions students with weaker coding skills who had good organizational skills began to take up roles as leaders and organizers. Many of the Trac tickets and wiki articles relevant to the development and coordination of the teams’ efforts were produced by these team members. This was particularly clear in the “B” team where hero programming efforts were foiled by the limited skill sets of the team members. Each team member had to contribute to the project or risk the project not being completed on schedule.

II. Results for Learning

I think that the approach of using myself as the Team Manager had a very positive influence on many of the students. I had evaluated several students as a TA in previous courses and these students demonstrated a remarkable improvement in the value that they added to their teams. Students were more able to provide guidance to team members and had a better understanding of how to collaborate to solve problems. Developers with weaker fundamental skills, particularly on the “B” team, rose to the challenges of organization and coordination, and worked closely with their teammates as they struggled through the project. The limited timeframe and resources, coupled with the relatively large scope of their assignments, resulted inevitably in crunch periods. Without effective “highly skilled programmers” on the “B” team, all team members were forced to work together closely, and as a result each gained not only a remarkable understanding of the underlying system, but also a very impressive work ethic and degree of team professionalism. This environment of close quarters and high stress appeared to teach the students to trust their teammates in a way that was not as evident on the “A” team, which benefited instead from the presence of a single very skilled programmer.

Despite this progress however, there are still several aspects that could be improved. The “A” team needed a larger work assignment to prevent any one student from being able to handle the majority of the development and design work. The strategy of assigning a substantial piece of work in order to induce students to work together well as a team worked very well for the “B” team, but the “A” team was able to handle their substantially larger module without leveraging their maximum potential as a team.

The result of this difference in team formation and operation became apparent when the final projects were evaluated. The “B” team adopted a strategy of minimizing the actual changes to the existing code except where

necessary, and maximized the amount of code that they could reuse. This strategy was sound given their limited skill set and lack of sufficient time to handle critical bugs (thus, a strong incentive to avoid creating any). In contrast, the “A” team criticized the quality of the code of the module that they were working on and decided to entirely replace the module which they were modifying, not reusing any of the tested code from the previous version. The “A” team, despite their criticism of the previous version which they were supposed to reproduce and extend, actually forgot to re-implement a critical feature which had been present in the previous version. Worse, the lack of this feature was explicitly pointed out in several code review meetings and yet was still omitted from the final product. It is my opinion that this oversight was the result of the asymmetric distribution of work present on this team and not actually a fault of any one member (however, it was possibly representative of a collective failing).

DISCUSSION

We can make several observations based on our experiences in teaching CPSC-240 Principles of Software Engineering. First, both the instructor and the TA share the common goal of preparing students for real-world software engineering. In addition, both instructor and TA desire students to learn how to work on a team. However, the emphasis placed on the balance between team and technical skills differs. The instructor places a greater emphasis on the technical skills related to how to develop software (e.g., requirements solicitation, design, test case construction, etc). This emphasis is likely due to two things. First, computing courses tend to put an emphasis on conveying technical knowledge and skills. Second, student progress in the technical aspects of a course is easier to measure (and grade), especially since the typical computing professor has little background in assessing soft skills such team dynamics.

In contrast, the TA puts greater emphasis on the development of team skills. The TA clearly sees the need for students to understand how to operate within and contribute to a team in a meaningful manner. This motivation stems from the observation that it is easier to teach technical skills than team skills. In addition, the team skills are more important to the long-term success of a student as students that cannot operate on a team will fail in their quest to become software engineers. In other words, it is easier to teach a student who already has an understanding of working in a team a technical skill (e.g., how to design using UML) than it is to teach a student with strong technical background how to work in a team.

In our reflection on teaching a project-based software engineering course we have identified several hindrances to learning. The first hindrance is due to the inherent nature of real-world software engineering which is complex and relies on a combination of learning from past experience, the correct selection and employment of development tools and approaches, and the proper fitting of development models

and practices to a team. It is this type of complexity that is difficult to manage within the confines of an academic course and also presents obstructions to student learning.

In addition, we have also identified that barriers to learning may be established in the early sequence of courses leading up to the software engineering project course. Mismatched teams in pair programming in introductory computing courses may lead to students developing a ‘hero programming’ mentality and do not foster an understanding of true teamwork or teamwork skills.

I. Future Improvements

Given an opportunity to run the course again, there are several aspects that we would like to change as well as several additional tactics that we would like to employ.

One essential requirement for preparing students entering CPSC-240 Principles of Software Engineering is to provide a better foundation of team experience in the courses leading up to the software engineering course. Clearly having students enter the course with a ‘hero programming’ mentality is a detriment to establishing good team dynamics. Suggested approaches to improving team skills include instruction and reflection on pair programming, feedback on team performance (as opposed to feedback on team artifacts), and instructor pairing of students based on student ability, specifically pairing students with similar abilities [12, 13].

Another approach that we intend to use in future offerings of CPSC-240 is to provide all teams with a sufficiently challenging project such that students have to rely on their teammates in order to solve the problem. Our experience has shown that projects that do not sufficiently cause students to extend their capabilities tend to result in individual rather than team efforts being used to complete the project.

In our original planning for the course, we discussed the possibility of swapping a member of each team with a member of the other team unexpectedly in the middle of the semester. This would force each team to deal with the challenge of bringing someone on board who was not familiar with the particular maintenance task as well as the challenge of losing a skilled and valued teammate. In doing so, the teams would be forced to rely on their documentation as a method of catching up their new team member in order to remain on schedule. Such a tactic would provide an excellent educational opportunity for the team members, as well as an interesting research opportunity for the instructors. Adding or changing members of a development team, especially on a late project, is often cited as one of the leading causes of delays in the development schedule. This effect can almost undoubtedly be mitigated through good software engineering practices. However, such an experiment is costly and risky to attempt in the real world, or even on institutional development teams due to the costs of schedule overruns. In a classroom setting, however, the net benefit of students experiencing changing team composition is actually positive.

It would also be very valuable and useful to develop an improved set of incentives to protect against free-riding on the development teams. Free-riding, where one or more student(s) does not make a significant contribution to the project, is a chronic issue which we have already discussed briefly as a method of reinforcing hero programming. It is also an issue with any established development team for practical purposes, as members are still able to avoid contributing. In many real world teams, these free-riding members are easily identified, and then cut as a cost savings measure. As it is impossible to fire members of teams in an academic environment, other artificial methods that invoke the same type of incentives must be conceived to help ensure higher levels of effort from all team members during the project. One possible solution would be a points-based system coupled with standard software engineering metric analytical tools which would draw data from Subversion and Trac. Systems such as Subversion and Trac provide a transparent view of the contributions made by individual members of a team.

CONCLUSION

The question of how best to convey software engineering knowledge and skills to undergraduate students is a difficult one. Students enrolling in a software engineering course are often interested in improving their abilities in developing complex software as well as exploring new technologies. One way to convey software engineering knowledge is to use case studies of existing systems and study the methods used to develop these systems, with a particular emphasis on post mortem analysis. While this approach might be effective in helping students to avoid many of the classic development pitfalls solved through good software engineering practices, it would not give them a practical understanding of how to add value to an active development team which must employ best practices to ensure quality software is produced and delivery schedules are met.

The approach that we took in CPSC-240 Principles of Software Engineering was to involve student teams directly in a project. This is a common approach for project-based software engineering courses, however we improved on this concept by employing a TA with real-world experience as a Team Manager. By forcing students to apply the practical software engineering knowledge that they are acquiring in the classroom in a controlled team centric environment, while retaining the significant value of case studies, post mortem analysis, and process techniques, we attempt to achieve a 'best of both worlds' approach. The use of the TA as a Team Manager allowed us to explore many of the challenges surrounding fostering team based approaches to problem solving in undergraduate educational environments.

REFERENCES

- [1] Hadfield, S. M., and Jensen, N. A., "Crafting a software engineering capstone project course," *Journal of Computing Sciences in Colleges*, Vol. 23, No. 1, Oct. 2007, pp. 190-197.
- [2] Alzamil, Z., "Towards an effective software engineering course project", *Proceedings of the 27th International Conference on Software Engineering*, May 2005, pp. 631-632.
- [3] Dascalu, S., M., Varol, Y. L., Harris, F. C., and Wesphal, B. T., "Computer science capstone course senior projects: from idea to prototype implementation", *Proceedings of 35th Annual IEEE Frontiers in Education Conference*, Oct. 2005, pp. S3J – 1-6.
- [4] Ellis, H. J. C., and Mitchell, R., "Self-grading in a project-based software engineering course", *Proceedings of the 17th Conference on Software Engineering Education and Training*, Mar. 2004, pp. 138-143.
- [5] Almstrum, V. L., Klappholz, D., and Modesitt, K., "Workshop on planning and executing real projects for real clients courses", *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* Mar. 2007, p. 582.
- [6] Toth, K., "Experiences with Open Source Software Engineering Tools", *IEEE Software*, Vol. 23, No. 1, Jan-Feb. 2006, pp. 44-52.
- [7] Venkatagiri, S., "Engineering the software requirements of nonprofits: a service-learning approach", *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 643-648.
- [8] Alshare, K. A., Sanders, D., Burris, E., and Sigman, S., "How do we manage student projects?: panel discussion", *Journal of Computing Sciences in Colleges*, Vol 22, No. 4, 2007, pp. 29-31.
- [9] van der Duim, L., Anersson, J., and Sinnema, M., "Good Practices for Educational Software Engineering Projects", *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 698-707.
- [10] Pfleeger, S. L., and Atlee, J. M., *Software Engineering Theory and Practice, 3rd Edition*, Prentice-Hall, 2006.
- [11] McConnell, S., *Professional Software Development: Shorter Schedules, Better Projects, Superior Products, Enhanced Careers*, Addison-Wesley, 2003.
- [12] McKinney, D. and Denton, L. F., "Developing Collaborative Skills Early in the CS Curriculum in a Laboratory Environment", *ACM SIGCSE Bulletin*, Vol. 38, No.1, 2006, pp. 138-142.
- [13] Braught, G., Eby, M.L., and Wahls, T., "The Effects of Pair-Programming on Individual Programming Skill", *Proceedings of the 38th SIGCSE*, Mar. 2008.

AUTHOR INFORMATION

Andrew Budd, BS in Computer Science and Economics Trinity College, Expected. MIT Sloan MBA candidate, class of 2010. C.E.O Adsci Engineerin, LLC.

Heidi J. C. Ellis, Assistant Professor, Trinity College, heidi.ellis@trincoll.edu