2010 Seventh International Conference on the Quality of Information and Communications Technology

Experiments with adding to the experience that can be acquired from software courses

Robert Dupuis
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Québec H3C 3P8
514-987-3000 ext. 3479
dupuis.robert@uqam.ca

Roger Champagne
École de technologie Supérieure
1100 rue Notre-Dame Ouest
Montréal, Québec H3C 1K3
514-396-8825
roger.champagne@etsmtl.ca

Alain April
École de Technologie Supérieure
1100 rue Notre-Dame Ouest
Montréal, Québec H3C 1K3
514 396-8682
alain.april@etsmtl.ca

Normand Séguin
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Québec H3C 3P8
514-987-3000 ext. 4345
seguin.normand@uqam.ca

ABSTRACT: This paper describes approaches used in two different software engineering courses, where the goal is to give students some experience in the major aspects of real world software projects. The first course is a capstone project course, part of an undergraduate short program in software engineering. The second course is a course on software maintenance and testing, part of a full undergraduate program on software engineering. Each course's content, general organization and student workflow is described. In the case of the capstone project course, graduate students are used as experts/clients in the context of a course in their own program. For the software maintenance and testing course, the emphasis is put on laboratory work. Both courses are considered to have succeeded with respect to the stated objectives. The positive aspects and major challenges with each course are also summarized.

1. INTRODUCTION

This paper revisits the theme of the capstone project and other experience-oriented software courses. We first describe a series of experiments performed on a project course given in the context of a short undergraduate program in software engineering at the University of Quebec in Montreal (UQAM), the primary objective of which is to help students gain experience in the fields of maintenance and quality. The original aspect of these experiments is that graduate students in software engineering were appointed as experts in either software quality or software project management to the class actually working on projects in these fields. The secondary objectives of the project course are to familiarize the students with the constraints and requirements of maintenance, i.e. working on a project that was started before their course began, and leaving an adequate product and its documentation to the groups that would follow theirs. We also describe another undergraduate course, offered at a different school, with the same objective of helping the students gain experience in the fields of maintenance and quality. This one was offered in a software engineering program at the École de technologie supérieure, an engineering school in Montreal.

The problems and limitations of trying to reproduce the professional context in a class are well known.

- The experience is limited to one or two semesters.
- It is difficult to teach quality skills. For instance, in programming courses, students practice unit testing and program testing, and not much more (Wikstrand, 2006)
- Students gain virtually no maintenance experience in regular courses.
- Most students have not had to be concerned with following a formal process in previous courses.
- Students have not had to suffer the consequences of insufficient testing of their software.

2. THE PARTNERSHIP EXPERIMENTS

The characteristics of the project course are the following. The undergraduate class is composed of students with little or no practical experience. Of those who have held computing jobs, virtually none has group software development experience. On average, they have taken between five and ten courses in computing, covering programming, databases, networks, and some basic software engineering.

The graduate groups came from a Master's degree program in software engineering. The students all had at least two years of professional experience in software engineering, most having closer to 10-15 years of experience. Although they had been involved in software projects, few had project management experience. Some also had quality assurance experience to some degree, but only a few had leadership experience in the field. Many more had only low-level experience in quality.

Consequently, the objectives of our experiments were to give undergraduate students some experience in the major aspects of real world software projects, and the graduate students an opportunity to play the roles of expert and staff member. The undergraduate students would learn to cope with:

- teammates they did not choose;
- fuzzy requirements to identify and select;
- a project already under way and deciding what parts to work on;
- preparation of adequate documentation at the end of their semester to help whoever would continue the project;
- a "boss" and business context to respect;

Conference'10, Month 1–2, 2010, City, State, Country.

978-0-7695-4241-6/10 $26.00 © 2010 IEEE
DOI 10.1109/QUATIC.2010.11

- expert staff members to deal with;
- some process to apply.

The projects have a real client (i.e. a person not related to the course), with whom the students had to discuss the mandate, and to whom they would try to deliver a useful product. Projects are submitted either by outside organizations or from within the university (labs, research projects, student bodies, etc.).

Some teams chose a new project. In that case, the students had to establish the needs of the client, general expectations of the software, and a first series of requirements. Other teams chose a project already started in a previous semester. In that case, the challenge was to understand what had already been done and what remained to be done. The students tended to realize quickly that documentation in this situation is very important.

Both categories had to properly document, at the end of the project, what they had done, and, more importantly, what they had not done. This included work that was started but only partially finished (the most difficult to explain on paper), or work that they had planned, but could not do at all.

The students had a "boss". The professor played the role of the owner of a small software firm. In that role, he could more easily explain the constraints of real-life contracts, and the requirements. This proved very useful in that it helped make the students aware of the importance artifacts such as time sheets. By explaining that they would not be paid without them, they accepted this chore, even though they were not going to be paid anyway! Before that aspect was added to the course, it was nearly impossible to get time sheets from the students.

The development process that is imposed is certainly a light one. We would qualify it as "opportunistic", since it varies according to the type of project: maintenance, new development, website, etc. Those who take on a project that is already under way are, in fact, conducting maintenance. Either they have to simply continue development or they are asked to make corrections to an existing system. In both cases, they are asked to list and analyze the requirements and order them according to importance and feasibility. Both those taking on a new development project and those building websites had to consider other issues, such as technological infrastructure, before looking at the ordering of work on the requirements themselves. Usually, we had to impress on them the need to allow sufficient time for these essential decisions to be made, because most are inclined to develop functionality above anything else.

In all cases, we had them develop a mandate and a project plan, and take all the time necessary to prepare documentation for the team that would follow them.

In terms of project control, we had them prepare a project schedule, regardless of the fact that their lack of experience made this very difficult, and control the evolution of their time budget and of the product itself. We consider this part of the exercise the most important experience for most of the students to acquire, both at the undergraduate and graduate levels.

We then add the standard objectives of such a course. The first of these is, of course, having students work on a team project! We don't let the students organize the teams themselves (Tilley et al., 2006). We identify as many people with experience as there will be teams, and ask them to choose among the projects. As we have mentioned, the students enrolled in this course typically have very little software project experience. At best, some have industry experience, but usually not of working on software teams. Consequently, any work experience that involved group activities, whether in software or not, was considered for this assignment. Also, of course, students have various skills and skill levels, and teams have to quickly organize themselves around those differences.

Then, the other students are asked to join the project of their choice, which usually requires some negotiation. We work this way to ensure that there is some experience in each team, which will be necessary if the mandate is not be completely fulfilled and the students have to work with new teammates, a more realistic experience.

The teams are made up of 3-4 people. We think this is the ideal number. With fewer than three, there is no communication problem to solve, and with more than four, it becomes more difficult to organize the participants, and the risk of having someone not doing his or her share increases.

By working on projects with real clients, the students have to determine the requirements and choose those they will try to implement. In entry-level programming courses, the professor usually establishes the specifications.

Among other contextual factors is the necessity for the students to combine addressing quality requirements and meeting demanding deadlines. Thus, they have to make realistic trade-offs. They also have to experiment with asynchronous communication with their teammates, their boss, the client, and expert/group, given that most of them are enrolled part-time, and in-person meetings which take place at most once or twice a week. We consider this realistic, since, in real organizations, not all the stakeholders (including teammates) are available at all times. In this context, 44 projects were conducted over a period of 8 semesters, some with staff appointed from graduate courses, some without.

The evaluation of the students includes the opinion of the client, the level at which the students respect the process imposed (two reports and presentations in class, time sheets, respect for the time budget, etc.), evaluation of the product, and an intra-team evaluation to determine which students, if any, did more or less than their share.

2.2 Roles of the graduate students

The first type of matching was with graduate students from a software project management course. Two roles were tried: PCO (Project Control Officer), and project planner and controller. As PCOs, the students would only audit and control the process followed by the groups. They had no authority whatsoever over the groups. Nevertheless, the developers knew that the PCOs reported to the same professor, and that their comments were important. One measure of success is that, in one term, we had one less PCO than teams, and that the orphan group complained at the end of the project that they could see the value the PCO had brought to the other teams.

As project managers, the graduate students did not have line authority over the teams, but were responsible for project planning and monitoring activities on top of their PCO activities. This was of greater interest, since these students were much more involved in the project than those who had only been PCOs, and consequently had much more to contribute to the success of the

project. In fact, the graduate students became members of the teams. In the first semester, where these project managers were used, we even had a prototype of a software project management tool to test. The tool proved to be too complex for the context, however, and the students considered it only as a burden and of very little benefit to them. We think that tuning such a tool for the context could, on the contrary, be a valuable addition to the course which could help students better control the time budget, for instance, and some other monitoring tasks.

Another major role assigned to the graduate students was that of quality assurance experts in a course on software quality. As such, they had to write a quality plan and a quality control procedure. During the term, they intervened to explain their plan and procedure to the developers. They were also in charge of helping the teams following the plans and procedures.

2.3 Major limitations

The projects themselves are very limited in scope. The time budget is 135 hours per student in the workbench class, and is variable for the graduate students, usually around 50 hours. This means that only relatively small projects can be tackled. Despite this limitation, we believe that the students at least have a taste of the most important difficulties they could encounter in a project.

Another major limitation is the fact that most of these students had day jobs, which means that it was challenging for them to organize meetings and engage in general communication. This is especially true for communication between undergraduate and graduate students, as they do not share a common class schedule. This difficulty was recognized both on a practical level and in terms of the consequences it had on the process and product. The students at least realized that, even in industry, communication is essential to the well-being of a project.

2.4 Lessons learned for the graduate students

The major lessons learned from the experiment by the Master's degree students are listed below:

First, they all agreed that this was a valuable experience. They felt that the difficulties they had to deal with were realistic.

The project control procedure described is, in fact, the result of lessons learned over the years. It became obvious that a somewhat formal process to track the progress of the product and of the budget was necessary, especially in this asynchronous context. It also was necessary to adjust the evaluation of the projects to give as many points to the process as to the product.

Another excellent suggestion, and practice, is to have groups share the templates they are using. Over the years, there has been a tendency to agree on relatively simple templates for all major documents: mandate, timesheets, etc., even though we always ask the groups to design part of the documents themselves. We believe this remains the best way for students to identify and deal with the major issues associated with each document. At the end of the terms, we would distribute the documents prepared by other teams and discuss the pros and cons of the various solutions proposed.

2.5 Conclusions

The answer to the most basic question, Did the projects succeed? is yes, on most grounds. The projects did deliver software that was considered useful by the clients. All of them agreed to continue development during the following semesters, and they now compete to have teams work for them.

On the teaching level, the experiments are considered a success as well. The students, both project and staff, did, in fact, accomplish some of the activities prescribed in project management and quality assurance. So, they must have gained some additional experience.

However, success is not automatic. The lessons learned from these experiments are the following:

- Insist on having the students continue with a project already under way, and not declare that the documentation and/or code is so bad that they have to start over from the beginning. The latter seems to be an easy solution and a way to go back to what they think they know best: develop from scratch;
- Insist on documentation quality, since, at the beginning of the semester, the teams have to understand the purpose of the project and the state in which it was left. So, towards the end of the semester, the teams must save some time for improving the documentation that was handed to them. It is a good idea to have them identify its shortcomings when they encounter them at the start. Also, try to limit deficiencies when their turn comes to prepare documentation for the following team.
- Another factor is the availability of the client, just as in real life. In fact, when students have difficulty reaching the client, it can be considered a plus in terms of their learning experience.
- Insist that the team become organized, and have them prepare a plan as soon as possible.
- We believe a major key to teaching success is strict enforcement of the time budget. The students each have 135 hours available, which means that they have to make choices and not simply keep working until the mandate is complete. They also have to make trade-offs with respect to documentation and quality activities.
- One of the best lessons they learn comes from the obligation to record all their hours and also to associate all work to an activity. At the end of the term, the students are amazed at the small number of hours actually available for coding. The amount of time devoted to communication and documentation always surprise s them.
- In fact, this scenario is very similar to most open source contributions: take a project on the fly, add something to it, and hope to leave it in a more advanced state in the end. It would even be a good idea to have some open source projects available to use in such courses. Consider having clear, high-quality guidelines or standards, and, ideally, groups to enforce them. In that case, students would have all the expected real-world conditions: real, useful projects, and quality requirements, both on results and process.

In conclusion, everybody involved becomes more excited than in any programming course.

3. THE MAINTENANCE AND TESTING COURSE

The other, comparable experiment was performed at an engineering school in an undergraduate software engineering program. The Software Maintenance and Testing course consists

of 13 weekly three-hour lectures and 12 weekly two-hour lab periods. The students gain their practical experience mainly from the lab periods, which is the focus of the rest of this section.

3.1 Maintenance portion of the course

After reading about teaching software maintenance and hands-on labs in work by Austin (2005), Postema et al. (2001), Slimick(1997), Andrews and Lufiyya(2000), and Allen et al. (2003), the course designers decided to use the same approach as Allen, who suggested that maintenance can be learned by having the students work on a program that is used by real clients. The students would work on maintenance for the first half of the course, and perform testing on modifications and corrections they had made to existing software in the second half of the course.

In the maintenance section of the course, there are 6 three-hour theory classes based on April's book on software maintenance management (April, 2008). This is complemented by 6 two-hour lab periods dedicated to experimenting with real-life maintenance situations. Each lab is divided in 3 two-week exercises. The overall objective of this portion of the course is to expose students to real-world conditions by having them to work on existing software that is poorly designed and undocumented. We would point out that the original developers are no longer available, and that defects must be corrected and new functionality added. Before this maintenance work can be undertaken, the software engineer must set up a suitable maintenance environment. Below, details of the content of each software maintenance lab exercise are presented.

In the first lab exercise (*installation and program understanding*), the students are told by their supervisor that a new client has some software and would like our maintenance section (composed of four students) to carry out the maintenance work. Unfortunately, the previous maintainer, A. Gile, has left the company and did not document the software, so only the source code is available. The students are given a copy of the source code of FinanceJ (http://sourceforge.net/projects/financej/), which we have modified so that it does not work properly. In addition, we provide a partially configured virtual machine containing helpful software, which will be used throughout the term, consisting of three individual tools: 1) a change request management system (http://trac.edgewall.org/); 2) a version control system (http://subversion.tigris.org/); and 3) a source code quality analyzer (http://qalab.sourceforge.net/). The students are told that they have to familiarize themselves with the maintenance environment and the software before they start any programming. Within the two-week assignment, they are asked to:

- make sure that the virtual machine provided by the client works and that the ticket tracking system (Trac) is functional;
- place the source code of FinanceJ under configuration management (version control in this instance) using Subversion;
- configure and test Trac:
  - define roles for a maintenance manager and maintainers;
  - configure permissions for each member of the maintenance team;
  - raise one ticket for each lab describing the work that has to be done;
  - configure milestones according to the 10-step maintenance workflow (see Appendix 1)
- configure and test QALab so that the client can extract knowledge from the source code. First assess the maintainability of FinanceJ with QALab. Place the resulting QALab reports in the Trac wiki to start documenting the source code. Identify the key maintainability problems of FinanceJ and raise perfective-type tickets in Trac;
- try to compile the software and identify the errors to be fixed. Raise as many corrective tickets as needed in the Trac system.

Finally, we tell the students that a quality assurance specialist (who is, in fact, the teaching assistant leading the lab sessions) will assess their work at the end of the two-week period.

In the second maintenance lab exercise (*reverse engineering*), students must reverse engineer the likely design (class and sequence diagram), as well as the use cases from the existing source code. For this we suggest that they install and use the Omondo tool (http://www.ejb3.org/). We also ask them to use the 'refactoring' option in Eclipse (www.eclipse.org) to improve any maintainability issues identified in the perfective maintenance tickets raised earlier. Their third task is to start fixing the defects and making improvements from the list of issues they identified during the previous lab exercise. Defect correction and perfective maintenance activities must be documented through Trac tickets and the maintenance workflow followed (see Appendix 1), and all artifacts must be placed under version control. Students are also asked to discuss how they have improved the maintainability of the code through refactoring.

In the third and last lab exercise on maintenance, new functionality must be added to FinanceJ. Students receive a number of change requests via e-mail from the client (who is in reality the teaching assistant) and are required to interact with the client solely via a Trac ticket. They must estimate the effort required to perform the change, and have the proposed solution and planned effort approved by the client. They must implement the change (on a separate SVN branch), update the project's documentation, perform a peer review within the team using a plug-in we added to the Trac configuration (http://trac-hacks.org/wiki/PeerReviewPlugin), and perform some ad hoc testing.

This approach enabled students to use the knowledge learned in class by:

- Experiencing the maintenance reality of large corporations, where the handover process is weak and maintainers are often left with only the source code of existing systems;
- Raising their awareness to the fact that a great deal of effort is required in setting up a suitable software maintenance environment and workflow before considering addressing any management or client requests;
- Experimenting with many types of maintenance requests (Corrective, Perfective, Adaptive) and understanding first-

hand their particular features in terms of workflow and toolset;
- Experimenting with redocumentation and refactoring techniques on an existing system, starting only with source code;
- Interesting them in the process maturity topics specific to software maintenance and how to gradually implement best practices in an unstructured and reactive environment.

3.2 Testing portion of the course

Following the maintenance part of the course, the focus switches to software testing. The first seven chapters of Burstein's book (2003) are covered in class. This book was chosen because it offers a good blend of basic testing concepts, rationale, techniques, and process, which is ideal for an engineering course. Interactive exercises are also performed in class on small examples from the main text and other sources, to ensure that students are taught the basic concepts and understand the various techniques enough to apply them on their own, which they will be asked to do in the lab exercises described below.

There are also 6 two-hour lab periods dedicated to the testing portion of the course, which is divided into 2 three-week exercises. The overall objective of this portion of the lab is for students to acquire minimal experience with basic black- and white-box testing techniques, by designing, implementing, executing, and reporting test results, and with the various levels of testing (unit, integration, system). They work on a different application for this portion of the course, because the one used for the maintenance portion did not allow us to highlight all the white-box testing criteria that we were aiming for (decision, condition, multiple condition, etc.). The same environment is used as for the maintenance portion of the course, the only notable additions to the toolset being the JUnit (http://www.junit.org/) testing framework and the Code Cover plug-in (http://codecover.org/) for Eclipse.

In the first testing lab exercise (*black-box testing*), students are initially handed a specification, screen capture, and public API of the application to be tested (a Gregorian calendar), and are asked to incrementally design unit, integration, and system black-box test cases. They have deliverables each week on which they receive feedback at the beginning of the following lab period. They are also provided with new information each week: the complete class diagram in the second week for the design of integration and system test cases, and the running application in the third week. They are only able to execute their tests in the last week. We work this way to ensure that they concentrate their efforts on test design (at the expected level, e.g. unit, integration, and system) and documentation for the first two weeks, and are not "distracted" by test coding and execution during this time. For each part, they must design the tests cases using the techniques shown in class, and document a test procedure (integration and system tests – unit testing is fully automated). They must report on any defects found, and their overall experience with the whole exercise.

In the second testing lab exercise (*white-box testing*), students are provided with the source code for the same calendar application, and a sample JUnit test class that contains sample tests, methods, and instructions on the expected format of their own test cases. For two methods in the main application class, they are required to draw the control flow graph (CFG), calculate McCabe's cyclomatic complexity (by hand – practical in this case because the application is small), identify the basis paths from the CFG, and, finally, design, code, and execute white-box unit tests that meet the following coverage criteria: 100% basis baths, 100% decisions and conditions, and 100% multiple conditions. They are required to meet each criterion with the smallest possible number of test cases, and to add test cases if they deem it necessary (justification required). Finally, they are asked to report on their general experience and, of course, report any defects found.

We conclude this section by summarizing the challenges for the testing part of the course. The course is given relatively early in the program (at the beginning of the second year), at which point the average student hasn't suffered the consequences of a lack of testing enough to fully appreciate its value (although the better students or those with more experience do). This could be remedied by offering the course later in the program, although that would require some non-trivial rearranging. The course also combines two related topics, each of which deserves its own dedicated full course. To make this change would require us to remove an existing course, which is not feasible at this time for practical reasons. The application used in this part of the course is different from that used in the maintenance part of the course. This is because the application chosen (first) for the maintenance part of the course does not allow us to expose the students to all the black- and white-box testing concepts and coverage criteria we aim to cover. Ideally, the same application would be used throughout the course. As is often the case, the application used is of academic size and scope. Given the short time allowed for the labs and the course positioning relatively early in the program, this is hard to avoid. Finally, as of this writing, testing concepts do not receive much attention in other courses in the program.

4. CONCLUSION

The two experiments prove that there are ways to provide students with some realistic experience. Lessons learned here seem to be that limited and well-defined projects, objectives, and process are probably keys to achieving some success in this regard.

Looking back at the limitations we mentioned in the introduction, we can say that these experiments, although still limited to one semester, help give students more realistic experience of working in a team and for a real client. They also show that it is possible, either in the project course or in the Maintenance and Testing course, to give students some realistic experience of maintenance. In both cases, we believe that open source software would be an appropriate source of software on which to work.

What remains is to find the time and the process to have students measure work beyond their time budget. We believe that the addition of quality measurement would be the next logical improvement to make to these courses.

Another improvement for the project courses would be to track risks. Students do prepare a risk assessment document at the beginning of the term, just after the mandate is received, but the tracking of risk has yet to be included. Moreover, since the students have so little experience, the risk assessment document is quite rudimentary. In our opinion, having them assess risk on an ongoing basis would be a very good addition to these courses.

In the Software Maintenance and Testing course experiment at the ÉTS, the focus was on familiarizing students with a realistic maintenance environment and process involving multiple

stakeholder roles, and introducing them to basic software testing techniques. We emphasize the "introduction" aspect, because one of the challenges of this particular course is that it is given relatively early in the Software Engineering program. Our experience thus far leads us to believe that students might assimilate the proposed concepts better if the course were given later in the program, especially considering that this is a coop program, and one or two coop terms might expose the students to a little more challenge, in the form of the absence of proper maintenance and/or testing activities in their work environment, and hence they would better appreciate the concepts taught in this course.

5. REFERENCES

[1] Allen, E., Cartwright, R., and Reis, C. 2003. Production Programming in the Classroom. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, Reno, NV, 9–93.

[2] Andrews, J. H. and Lutfiyya, H. L. 2000. Experiences with a software maintenance project course. IEEE Transactions on Educations, 43, 4, 383–388.

[3] April, A. and Abran, A. 2008. Software Maintenance Management: Evaluation and Continuous Improvement. Wiley-IEEE Computer Society, 314.

[4] Austin, M. A., III and Samadzadeh, M. H. 2005. Software comprehension/maintenance: An introductory course. 18th International Conference on Systems Engineering (ICSEng 2005), 414–419.

[5] Burnstein, I. 2003. Practical Software Testing, Springer, 400.

[6] De Koenigsberg, G. 2008. How Successful Open Source Projects Work, and How and Why to Introduce Students to the Open Source World. In Proceedings of the 21th Conference on Software Engineering Education & Training (CSEE&T ’08).

[7] Fornaro, R. J., Heil, M. R., and Tharp, A. L. 2006. What Clients Want – What Students Do: Reflections on Ten Years of Sponsored Senior Design Projects. In Proceedings of the 19th Conference on Software Engineering Education & Training (CSEE&T ’06).

[8] Padua, W. 2009. Using Quality Audits to Assess Software Course Projects. In Proceedings of the 22th Conference on Software Engineering Education & Training (CSEE&T ’09).

[9] Postema, M., Miller, J., and Dick, M. 2001. Including Practical Software Evolution in Software Engineering Education. In Proceedings of the 14th Conference on Software Engineering Education and Training, Charlotte, SC, 127–135.

[10] Slimick, J. 1997. An undergraduate course in software maintenance and enhancement. Tenth Conference on Software Engineering Education & Training, 61–73.

[11] Tilley, S., Huang, S., Wong, K., and Smith, S. 2006. Report from the 2nd International Workshop on Software Engineering Course Projects (SWECP 2005). In Proceedings of the 19th Conference on Software Engineering Education & Training (CSEE&T ’06).

[12] Wikstrand, G. and Börstler, J. 2006. Success Factors for Team Project Courses. In Proceedings of the 19th Conference on Software Engineering Education & Training (CSEE&T ’06).