Teaching an Advanced Design, Team-oriented Software Project Course

Stan Jarzabek and Pin-Kwang Eng
Department of Computer Science, School of Computing
National University of Singapore
{stan,engpk}@comp.nus.edu.sg

Abstract

Students learn about design principles and "best practices" in many courses. However, small scale assignments do not give enough opportunity for students to appreciate the value of software design principles or even to learn how to apply principles in practice. To fill the gap between theoretical and experiential knowledge, we introduced a team-based project course focused on design and implementation phases of the software development lifecycle. We teach design principles and team work in problem-based way, through architectural concepts and iterative development process. The product students build must meet stated quality requirements in terms of reliability, reusability and documentation. We trust this kind of the course is essential in curricula as it allows students better absorb knowledge learned in other software engineering courses. Such course also plays a role in better preparing students for industrial work. We describe a teaching method, course infrastructure and lessons learned over three years of teaching of our course. Based on experiences, we postulate and motivate the need for teaching at least two project courses in undergraduate curricula, one dealing with design and process issues, and the other focused on unstable requirements.

1. Introduction

The need for teaching project-oriented courses in engineering disciplines has been widely recognized [7]. The motivation for teaching software engineering project courses is to better prepare students for industry projects. The ultimate goal is that fresh graduates not only can faster contribute to industrial projects, but can also play the role of agents transferring principles and "best practices" into industry. Project courses are key means to achieve this ambitious goal.

Project courses emphasize team work, systematic development based on "best practices", a rigorous development process, communication skills and other qualities that matter in industrial projects but cannot be effectively taught in assignment-based courses or acquired in industrial attachments. To meet different expectations, a number of models for project courses have been proposed, ranging from projects proposed/supervised by faculty members, to projects done as part of a software engineering course, and to stand-alone project courses. In this paper, we focus on stand-alone team-based project courses taught in the framework of a university, as opposed to industrial attachments. Though there is still much variation among such courses, we can distinguish two major trends: Courses that emphasize problems of fuzzy and changing requirements, and courses that focus on the design and rigorous development process.

Unstable requirements and software design are not only the two hallmarks of software development, but also hard and wicked problems [2], that are difficult to teach in the frame of a single course. Therefore, in courses that attempt to expose students to the reality of fuzzy and changing requirements, the development phase tends to be less structured and rigorous.

On the other hand, courses that emphasize design principles, "best practices" and a rigorous process are usually well-structured, but use a relatively stable set of requirements. We believe that ideally students should take at least two project courses in the undergraduate education, one dealing with design and process issues, and the other focused on unstable requirements. To observe and understand the role of design principles, the first course should limit factors that make application of principles particularly difficult. In this paper, we describe a project course that emphasizes the learning of design principles and "best practices" in preparation for the second course that deals with fuzzy/unstable requirements.

Students learn about design principles and "best practices" in many courses. However, small scale assignments do not give enough opportunity for students to appreciate the value of software design principles or even to learn how to apply principles in practice. To fill the gap between theoretical and experiential knowledge, we introduced a team-based project course focused on design and implementation phases of the software development lifecycle. We teach design principles and team work in problem-based way, through architectural concepts and iterative development process. In the paper, we describe a teaching method, course infrastructure and lessons learned over three years of teaching of our course.

The paper is organized as follows: In the next two sections, we discuss the rationale and overview of our project course. A tool environment is described in Section 4. In the remaining sections, we discuss evaluation of students' projects, provide course statistics and assess results so far.

2. Background

At the National University of Singapore (NUS), we experimented with various approaches to teaching project courses. Although our students participated in the industrial attachment program and did projects proposed by faculty members, industry surveys were consistently signaling the same problems related to weak development and communication skills, and generally slow start in industrial projects. When exposed to real world pressures, our students did not know how to take advantage of what they had learned. Students tended to perceive principles as obstacles rather than tools that can help them better complete the project work. Assignments in programming courses involve small programs. For example, if 1-2 students working closely together can develop a program in a couple of days and the assignment requires them to define an architecture and interfaces, students will do so just to satisfy assignment requirements, not because there is a real need for those artifacts in the context of assignment's practical goals. The value of most of the principles can be only appreciated when it comes to large programs, developed by teams, when we need to understand different program parts separately, split the job into tasks that are completed in a fairly independent way and develop a product incrementally rather than in one shot.

Industrial attachments offer an invaluable experience, but not all the companies expose students to "best practices" or let student teams go through the whole development lifecycle. University project courses organized in ad hoc way carry heavy workload for both instructors and students, and, at the end, students do not learn enough to make the effort worthwhile. Projects tend to be focused on specific research interests of a supervisor and only few supervisors will be really concerned with rigorous software development based on "best practices".

With the above experiences, we took a fresh look on what kind of practical experience would be most beneficial for our students. Given the difficulties of teaching a rigorous development in the frame of a project that also deals with fuzzy/unstable requirements, we opted for two courses. The first course is based on a business application, with emphasis on

requirement elicitation, with fuzzy and incomplete requirements given to students, and unexpected changes of requirements. Students spend much time on prototyping to validate user requirements, design the user interface and the database. The second course, CS3215, focuses on advanced design and rigorous process. The application domain guarantees challenging design problems, emphasizes the role of software architecture and component interfaces, and involves complex data structures and algorithms. The problem is selected and scoped in such a way that students cannot meet the project goals without applying software engineering principles and "best practices" that we recommend to them. The course aims at enhancing skills related to communication and product quality in view of those challenges. In the remaining part of the paper, we discuss CS3215 in details.

3. An overview of CS3215

The project is delivered in five phases, comprising requirements analysis and architecture design, followed by three development iterations. Iterative development (based on Unified Process [3]) and software architecture are the key concepts shaping the project and the key means to achieve teaching goals of our course.

The project course starts with 10 lectures during which a chief instructor motivates students, clarifies course objectives, explains the programming problem, project methodology and the development process. Students do the project in teams of six students. Teams are further divided into two groups of three students. Each team is assigned a one-hour slot per week for consultation with a supervisor. Supervisors – a chief instructor and teaching assistants - are intimately familiar with all the technical aspects of the project, and share a common vision of the project course objectives.

Students spend two weeks on problem analysis and two weeks on architectural specifications. At the same time, they develop a throw-away prototype. A programming problem – a software tool called Static Program Analyzer, SPA for short – has been carefully selected to allow the two groups to work on the two subsystems in a fairly independent way. We explain subsystem-level SPA decomposition to the students, and students' task is to follow up with component-level decomposition and specifications of major component interfaces. Without proper definition of the interfaces it is virtually impossible to integrate the two subsystems. At the same time, the interface is complex enough so that some changes and refinements of interfaces are inevitable during development iterations. To cope with that, students must work together during architecture design and also meet on regular basis during development. To effectively communicate in a team, students learn how to schedule meetings and how to write documentation that is understandable to other team members.

Students develop the project in iterations. The iterative development helps students tackle project difficulties one by one, applying principles of separation of concerns, abstraction and refinement.

Program reliability is emphasized throughout the project. High reliability is achieved by reviews and comprehensive testing. We expect program reliability close to industry standards. Students are advised to allocate enough time for test planning and testing, to make unit testing an integral part of development and to do integration testing and system testing often, at least at the end of each of the development iterations.

We give students a Project Handbook that includes problem description, compendium of recommended software engineering practices for the project, sample solutions illustrating how we expect them to approach design problems and technical tips. At the end of the course, students write a report, present their solutions and their programs are tested for errors with an auto-tester running some 250 test cases.

We developed a Project Infrastructure for efficient delivery of the project course. The Project Infrastructure consists of handouts (a Project Handbook, assignments, and tools' documentation), lecture notes, and tool environment (configuration management, testing and modeling tools).

4. A tool environment

One of the key components of our Project Infrastructure is a tool environment consisting of a set of open source testing, source code control, build and documentation tools.

4.1. Introducing the tools to our students

We wrote short tutorials to help students install and use the tools. A standard documentation for the tools targets at a wide range of audiences, application domains and computing environments. Our tutorials, on the other hand, highlight tools from the perspective of our project, complementing rather than replacing the standard tool documentation, and providing our students with a quick start for tool usage.

We also introduced compulsory training lab sessions, conducted before the teams started their actual development work. These lab sessions aimed to motivate the students to use tools and to shorten the tool learning curve. Overall, we found this approach effective as more teams adopt the tools for their development work. We also observe that some teams will even go to the extent of developing additional tools for their projects. Such teams show a high level of initiative and desire to produce a high quality product and we reward them for such effort.

4.2. Recommended tools for the project course

Students use four categories of tools for the project, namely testing tools, source code control tools, build tools and documentation tools. The implementation language for the project is either Java or C++, for which there are plenty of good tools in each category.

Testing tools. Reliability is one of the key qualities we look for in the project course and students do much testing to meet the reliability requirements. Students use JUnit [11] for Java and CppUnit [9] for C++. Even though we do not introduce tools for integration or system testing, many teams come up with their own integration and system testing tools.

Source code control systems. Students use the Concurrent Version System (CVS) [10] to manage multiple versions of the source code created during the project. CVS provides record keeping (i.e., revision control) as well as facilitates collaboration. We got very positive feedback from teams who have adopted CVS. To them, CVS has become an indispensable tool, one they know they cannot do without for programming in the large.

Build tools. Build tools are used to automate the build [2], test and deployment process in a large project. For Java, students use Apache Ant [8]. Ant is easily extensible using Java classes and can automate the running of JUnit test suites. This is especially important for regression testing when all the test suites have to be re-run after making some changes to the system. For C++, students use the *make* utility [12] which is widely used in many C/C++ projects.

Other tools. For documentation, students use Javadoc and Doxygen [13] for program documentation in Java and C++, respectively, and ArgoUML [14] for drawing UML

diagrams. We also introduce additional tools such as graphical toolkits, advanced data structures and analysis tools [15]. However, we discourage the use of Integrated Development Environments (IDEs) unless all the team members are familiar with the IDEs they want to use. This is because we find that students end up spending more time learning and solving problems related to the IDE than problems of the project.

Before we end this section, we would like to comment on plagiarism which is a great concern in project courses. We adopted a clone detection tool CCFinder [5] to find cases of copied solutions. CCFinder is a token-based tool that can work with different programming languages. CCFinder is highly parameterized so that it can be customized to the clone detection task at hand. A visual interface called Gemini displays clone statistics in graphical form which allowed us to eliminate false positive clones – that is clones not indicating plagiarism - such as library classes. We think that availability of automated clone detection techniques can deter plagiarism of code to a great extent. It cannot prevent students from copying analysis and design solution, though. However, we notice that such cases do not happen often and they immediately show during consultations.

5. Evaluation of students' projects

Students' projects are evaluated based on the scope of the program functionality implemented, programs' quality attributes (reliability, reusability, extensibility and the efficiency of a query evaluation strategy), and the quality of project documentation.

At the end of the project course, each team is given one hour to present their work and to complete a final system testing. One test run, consisting of 200 to 300 test cases covering a large set of functionalities, is executed for each team. Important information such as failed cases, exceptions and timeouts (when the time taken to evaluate a query does not meet the time limit we set) are captured. Any test cases that failed are then re-run and the results are verified manually. Students are allowed to give explanations of what went wrong if they know the reason. Hence, the final testing will give us an objective score on the reliability of the students' programs.

To facilitate the final system testing, we developed a tool called *AutoTester* to automate testing of students' programs. The AutoTester works with students' programs according to a client/server architecture concept: The AutoTester - the server - reads a set of test cases (program queries in our case) and then repeatedly sends queries to a student's SPA program - the client. The SPA evaluates the query and returns the results back to the AutoTester for verification. To minimize the problems that might occur during the final testing, a trial run is conducted for each team prior to the final testing. This not only helps to familiarize the students with the testing procedures but also highlights any problems the team has when using the AutoTester.

6. Project course statistics

We present some statistics to give a reader a better idea of our course. We gathered statistics from two offerings of the course, in which students used Java and C++, respectively.

6.1. Errors in implementation

Figure 1 and Figure 2 illustrate the distribution of errors found in the final testing. An error is either a unique failed case, an exception or a timeout. For each team, we show the number of each type of errors encountered during the final testing.

From the figures, we can see that in general, most teams manage to keep their total number of errors below 10. We believe that our relentless emphasis on reliability

manage to drive home an important point to our students – that testing is important in any large scale software development. We also make the following observations. First, we find that in most teams, at least one unique failed case is attributed to the team's misunderstanding of the specification. Some teams purposely imposed their own restrictions on the project to reduce its complexity, failing to meet our project requirements. Second, programs that are implemented in Java have fewer total errors because Java is frequently used for programming assignments in most courses in our university and hence the students are more proficient in using Java than C++. Finally, C++ programs tend to have many timeouts but few exceptions. We believe that lack of error checking mechanisms in C++ forced our students to be more conscious about error/exception handling and recovery when implementing in C++, resulting in fewer exceptions. However, as most of our students are exposed to C++ for the first time, they tend to code in a very inefficient way, resulting in more timeouts.
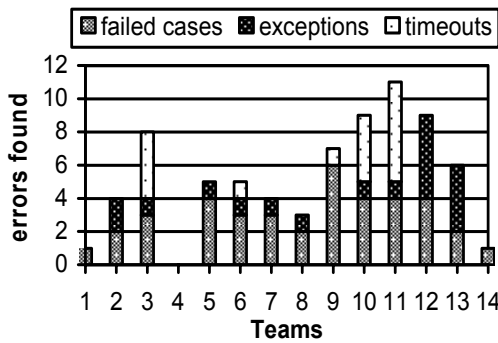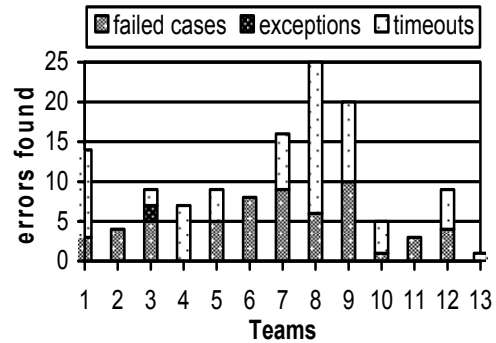

Figure 1. Distribution of errors (Java programs)

Figure 2. Distribution of errors (C++ programs)

6.2. Program size

Figure 3 and Figure 4 illustrate the distribution of program sizes in thousands of lines of code (KLOC). The mean size is 8.9 KLOC and the standard deviation is 3.1 KLOC for Java programs. For C++ programs, the mean size is 9.7 while the standard deviation is 2.8 KLOC.
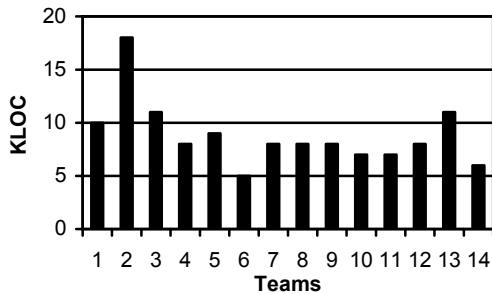

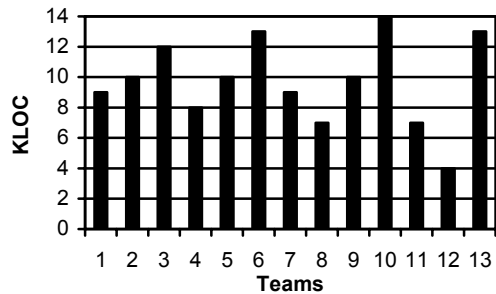**Figure 3. Program size (Java programs)**

Figure 4. Program size (C++ programs)

Sometimes, by inspecting program metrics, we could spot cases of unusually good or bad design. Notice that the Java program developed by the second team was much larger than the rest. Upon investigation, we found that it was an instance of a bad programming practice: Rather than developing a few generic methods to handle the various types of queries, this team created one method for each query type. As the number of query types is large, this

resulted in 132 methods (totaling 10 KLOC) for just one of the classes alone, with much code duplication across methods. Finally, we observe that the C++ programs are generally larger than the Java programs. This is due to the fact that C++ programs need extra code to be written to perform runtime checks, but also because students still lack enough C++ skills to come up with short and elegant detailed design solutions.

7. Results so far

We have been offering the CS3215 project course in both terms for last three years, to 80-130 Computer Science students each term. We see a difference in the way students deal with team-based software development at the end of the project as compared to their approach at the beginning of the course. Students appreciate the role of software architecture, learn how to communicate in terms of interfaces, how to split the project work and document the work products in a clear way, understandable to other teammates. Most of the teams become very much involved in the project and motivated to work hard in order to deliver a quality product. Though in this course we make little effort to emulate the real world, we believe students learn some essential skills that help them to deal with real world project challenges in a systematic rather than chaotic way.

Improving communication skills is the major concern and the focal point in our project course. The ability to express solutions at the concept level, to properly document design, code, test plans and other program artifacts, to describe interfaces and to use assertions - are all related to communication skills. So is the ability to prepare and conduct team meetings in an effective way. Communication skills integrate essential human and technical aspects of software development. As students do not have enough opportunity to develop communication skills in assignment-based courses, it is the major challenge for the project course to address this issue. We believe our course gives students ample opportunity to experiment with a wide range of techniques related to mastering communication skills.

The Project Infrastructure plays a critical role in achieving teaching goals. It helps students learn about software tools in short time and helps instructors in evaluating program solutions. At many universities, project courses are offered to a large population of students. A common problem is to find enough faculty members and teaching assistants qualified for the job. It takes a long time before teaching assistants can provide useful advice to student teams. Our approach to teaching the project course and the Project Infrastructure alleviates these problems. The Project Handbook and Infrastructure communicate to the students "what and how" of the project. They are a repository of project information that is shared among instructors, shortening the learning time before the new instructors can effectively advice student teams. The Project Handbook and Infrastructure can be used "as is" or customized to account for specific goals, student audiences and specific qualities deemed important in a given offering of a project course.

In summary, we found our approach to teaching a project course effective in (1) enhancing students' skills related to team work, communication and applying principles and "best practices" in-the-large, and (2) in dealing with practical problems related to teaching a project course for a large population of students.

8. Conclusion

If the main role of universities is to teach fundamental concepts then the role of project courses is to teach how to apply those concepts in large-scale team-based software development. To fill the gap between theoretical and experiential software engineering knowledge, we introduced a team-based project course focused on design and implementation phases of the software development lifecycle. We adopted a rather formal approach to

teaching this new project course. We teach design principles and team work in problem-based way, through architectural concepts and iterative development process. We conduct the project course in such a way that it is impossible for students to achieve the project goals unless they follow the path of "best practices" we are showing to them. In the paper, we described a teaching method, course infrastructure and lessons learned over three years of teaching of our course.

Unstable requirements and software design are not only the two hallmarks of software development, but also hard and wicked problems [2]. They are inter-related in subtle ways which, however, may have profound impact on the project success or failure: On one hand, tensions arise when rigid and premature design structures become restrictive to frequent and vast changes triggered by evolving requirements. On the other hand, ad hoc and poor design hinders development and maintenance of complex systems. We believe that, ideally, students should take at least two project courses in the undergraduate education, one dealing with design and process issues, and the other focused on unstable requirements. To observe and understand the role of design principles, the first course should limit factors that make application of principles particularly difficult. Having understood how principles work in a relatively "clear water" situation, in the second project course, we can more realistically expect students to work with unstable requirements (e.g., by applying agile methods [1]), without steering their projects into the chaos, with little educational value gained at the end.

9. Acknowledgments

References

[1] AgileAlliance http://www.agilealliance.org/home

[2] Armarego, J. Advanced Software Design: a Case Study in Problem-based Learning," Proc. 15th Conf. on Software Engineering Education and Training, CSEET'02, Covington, USA, Feb. 2002,

[3] Jacobson, I., Booch, G. and Rumbaugh, J. The Unified Software Development Process, Addison-Wesley, 1999

[4] Jarzabek, S. (editor), "Teaching Software Project Courses" (special issue, 13 papers), Forum for Advancing Software Engineering Education, Vol 11, No 6, http://www.cs.ttu.edu/fase/v11n06.txt, June 2001

[5] Kamiya, T., Kusumoto, S., and Inoue, K. "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code", IEEE Trans. Software Engineering, 2002, 28(7): pp. 654-670

[6] Robillard, P. "Teaching Software Engineering through a Project-Oriented Course," Proc. Conf. on Software Engineering Education, CSEE'96, 1996, pp. 85-94

[7] Panel Discussion "Learning Objectives for Engineering Education Laboratories", Session F1D, Proc. Frontiers in Education, p. F1D-1, Nov 2002

[8] Apache Ant. http://ant.apache.org

[9] CppUnit. http://sourceforge.net/projects/cppunit.

[10] Concurrent Version System. http://www.cvshome.org.

[11] JUnit. http://www.junit.org.

[12] GNU make. http://www.gnu.org/software/make.

[13] Doxygen. http://www.stack.nl/~dimitri/doxygen.

[14] ArgoUML. http://argouml.tigris.org.

[15] JDepend. www.clarkware.com/software/JDepend.html