A Hybrid Design Methodology for an Introductory Software Engineering Course with Integrated Mobile Application Development

Vignesh Subbian, University of Cincinnati

Vignesh Subbian is an instructor/teaching assistant in the Department of Electrical Engineering and Computing Systems at the University of Cincinnati. His research interests include embedded computing systems, medical device design and development, point-of-care technologies for neurological care, and engineering education.

Dr. Carla C. Purdy, University of Cincinnati

Carla Purdy is an associate professor in the School of Electrical Engineering and Computing Systems, College of Engineering and Applied Science, at the University of Cincinnati and an affiliate faculty member in UC's Department of Women's, Gender, and Sexuality Studies. She received her Ph.D. in Mathematics from the University of Illinois in 1975 and her PhD. in Computer Science from Texas A&M University in 1986. She is the head of UC's B.S. in Computer Engineering Program and the coordinator of the Preparing Future Faculty in Engineering Program. Her research interests include embedded systems and VLSI, intelligent embedded systems, software and systems engineering, computational biology and synthetic biology, agent based modeling and simulation, mentoring, and diversity in science and engineering.

A Hybrid Design Methodology for an Introductory Software Engineering Course with Integrated Mobile Application Development

Introduction

This paper discusses an experimental version of a core undergraduate software engineering course at the University of Cincinnati (UC). EECE 3093C – Software Engineering is a 4-credit hour undergraduate course with an integrated laboratory component. It is a required course for all computer science and computer engineering students. Traditionally, this course consisted of in-class lectures, along with laboratory projects that required students to develop software for a serious game based on a discrete-event simulation model using Java. The course design process was built on the waterfall model, integrated with important concepts from extreme programming (XP), including test-driven development using three levels of design and testing (system, black box, and glass box) and an onsite customer. When UC recently converted their academic calendar from quarters (10 weeks) to semesters (14 weeks), the additional instruction time provided an opportunity to revisit and expand the design process model of the course. In addition to the existing features of the course that allow effective instruction in contemporary software engineering principles, the experimental version of the course incorporated the following variations:

1. The laboratory project now involves open-source mobile application development;
2. The hybrid design methodology (waterfall and XP) is further explored by incorporating two or more development cycles into the project, while additional classroom activities further understanding of connections between the development process and application needs;
3. Five active-learning sessions are included to enable reflection on past co-operative education or internship experiences and relate them to classroom learning. The objective of this novel pedagogical strategy, which we call *UnLecture*, is to bridge the gap between software engineering practice and computing education.

Background and Review

Software engineering, since its inception as a discipline in the late 1960s[1], is continuing to change and evolve in concert with advancements in computing hardware and software-intensive systems. Hence, from an educational standpoint, it is important to frequently assess software engineering practices not only to refresh curricular material, but also to strike a balance between "aging practices" and "timeless principles" in software engineering instruction[2]. To this end, the course EECE 3093, that has been taught for over a decade at UC, has always strived to maintain a balance between traditional and contemporary concepts and techniques. The primary ABET student outcomes (a-k) addressed in this course are:

1. Comprehend software development life cycle models, and project planning and organization, for both traditional and distributed projects (a, g).
2. Understand how to develop specifications, design, and test code for a set of software requirements and how to measure the quality of software developed and of the development process itself (a, e).
3. Use team-building skills to work with the student's team to plan, design, implement, test, and develop a mobile application (a, c, d, e, g, k).
4. Comprehend formal software engineering methods (a, e).
5. Apply principles of the ACM/IEEE Software Engineering Code of Ethics to class work (d, f).

A few supplementary student learning outcomes are as follows:

6. Identify and relate real-world/cooperative education experiences to coursework, and reflect on the connection between classroom learning and software engineering practice (i).
7. Comprehend global software engineering concepts and challenges (a, h).

Course Design

Lectures: The course material primarily focuses on the first five Knowledge Areas (KAs) of the Software Engineering Body of Knowledge (SWEBOK)[3]: software requirements, design, construction, testing, and maintenance. While the remaining KAs are covered in a newly developed upper-level course (CS 6028- Large Scale Software Engineering), a brief introduction is given to a few topics such as software quality and software engineering methods. In addition to these KAs, other topics in the course include software engineering ethics and global software engineering. A detailed list of in-class lecture topics is shown in Table I. Topics with active learning component (*UnLecture*) are also marked in Table I.

Table I List of Course Topics (Summer 2013)

ID	In-class Lecture Topic	# Lectures	*UnLecture*
1	Overview of Software Engineering Principles	1	
2	IEEE/ACM Software Engineering Code of Ethics	1	✓
3	Software Development Life Cycle (SDLC) models	3	
4	Project Planning and Management	1	✓
5	Software Requirements Analysis	3	✓
6	Software Design and Modeling	3	✓
7	Object-Oriented Programming (Core Concepts)	2	✓
8	Mobile and UI programming, APIs	5	
9	Software Design Patterns	2	
10	Software Implementation, Re-use, Best Practices	3	
11	Software Testing and Quality	6	✓
12	Software Deployment, and Maintenance	3	✓
13	Software Delivery, Business/Legal Aspects	2	✓
14	Formal Software Engineering Methods	2	
15	Global Software Engineering	1	✓

UnLectures: The undergraduate engineering programs at UC include a strong cooperative education (co-op) program in which students work in industry every other academic semester, completing five co-op rotations upon graduation. An *UnLecture* is a participatory session designed to “tap” the knowledge and expertise that students gain through cooperative education and utilize this knowledge and expertise to facilitate meaningful discussions related to the course topics. Each *UnLecture* involves a reflective writing component before and after a “themed” active-learning session. Five *UnLectures* based on the following themes were introduced in the Summer 2013 class: (1) project management, (2) design and requirement analysis, (3) software implementation practices in the industry, (4) software testing and maintenance, and (5) software engineering ethics and technology/patent wars. These five sessions jointly cover nine course topics as shown in Table I. More details on the pedagogical model of the technique, *UnLecture* logistics, and related findings are elaborated on in a separate paper[4].

Laboratory project: Students are required to design, develop, and test an Android® mobile application using a hybrid design process model. This design methodology, as mentioned earlier, is based on the traditional waterfall model, integrated with important XP principles such as Test-Driven Development (TDD), small releases, and on-site customer. During the Summer 2013 offering of this course, several other XP principles were also incorporated into one or more releases. The benefits and pitfalls of using these XP principles in a classroom setting are discussed in a later section. A total of nine small releases were executed, with each release emphasizing a waterfall and/or XP principle. Table II shows the list of releases along with the timeline and emphasis for each release. Although Table II implies multiple incremental releases, the two pre-alpha releases are in fact not software releases but are actually outcomes of the requirements and design phases in a traditional waterfall model, and subsequent releases are biased towards an XP model through incremental application development.

Table II Laboratory Project Release Cycle

Release Name and Version #	Timeline	Emphasis (Summer 2013)
Zero-feature Release	Week 2	System metaphor, software reference documentation
Pre-alpha 0.1	Week 4	Requirements analysis (vs simple design)
Pre-alpha 0.2	Week 6	Object-oriented modeling, coding standards
Alpha 0.1	Week 8 Week 9 (Design Reviews)	Test-Driven Development (TDD), continuous integration, pair programming
Alpha 0.2	Week 10	The planning game, TDD, 40-hour week
Beta 0.1	Week 11	TDD, code reviews
Beta 0.2	Week 12	Refactoring, on-site customer
Release Candidate (RC)	Week 13	Coding standards, refactoring
Release-To-Manufacturing (RTM)	Week 14	On-site customer, collective code ownership

Migrating the laboratory assignments and project from web-based/general-purpose application development to mobile applications obviously comes with the cost of changing and setting up the new development environment and relevant tool support. The Android Software Development

Kit (SDK), however, is relatively straightforward to set up, and comes with complete support for application development, testing, and debugging, a mobile device emulator, and extensive documentation. The original version of the course required students to build applications in the Java programming language. Since Android applications are also written in Java, the only overhead was adapting to the Android SDK. Furthermore, given the popularity of mobile and tablet devices, students were generally enthusiastic about learning to build such applications.

Student Assessment: Exams, individual assignments, and participation in *UnLectures* and associated reflective writing constitute 50% of the grade. The laboratory project constitutes the remaining 50% of the course grade, and students are assessed based on both individual contribution and team performance. Every release (see Table II) is graded based on their documentation, design reviews, code correctness, and demonstration. Design reviews are delivered in the form of in-class oral presentations. Code correctness is assessed by running various test cases written as a part of TDD. Each team is also required to maintain a productivity chart to track progress and guide development plans for upcoming releases. Releases are held biweekly during the first half of the term to allow for students to become acclimatized to the development environment and the design process model of the course. After the midterm, releases are held on a weekly basis. Thus, the release cycle, in effect, emulates a waterfall model at the beginning by providing sufficient time to carry out detailed requirements analysis and design, and tends to shift towards an agile/XP model once the implementation has begun.

Benefits and Pitfalls of using XP Principles in a Classroom Setting

With the intention of evaluating our hybrid design methodology in a classroom, a subset of the 12 generally accepted XP practices[5] was integrated with some aspects of the traditional waterfall model. The extent of conformance and the feasibility of each practice, as observed in the Summer 2013 laboratory, are as follows.

Simple design and the planning game: A key difference between the waterfall and XP models is that the waterfall model would tend to gather the requirements and define the specifications at the beginning, whereas XP starts with the "simplest possible" design and then builds/modifies the system gradually, with each iteration starting with user stories about what should be added or modified next. User stories and iteration planning, collectively referred to as the planning game, are not always suitable for applications where safety and security are major concerns, and are not part of the development process at many of the local companies where our students co-op and often become full-time employees. Alternatively, detailed requirements analysis and object-oriented modeling using UML[6] is used to derive specifications early in the development process.

Test-Driven Development (TDD) and small releases: Once the design and requirements analysis phases are complete, implementation and testing are done incrementally in the form of small releases. The practice of "small releases", as shown in Table II, was perceived to be extremely useful for both the instructor and student teams in evaluating and keeping track of progress in system development. TDD in the hybrid methodology is defined as follows: a tester first defines boundary tests for the modules to be added or modified by treating them as "black boxes" and then hands them off to a developer. The developer would then write the code, and also "white box" tests to make sure that their code does what it is supposed to. The developer would also need to make sure that inputs and outputs are consistent with the specs they were given, to facilitate integration. After a module passes both white and black box tests, the team qualifies it to be a candidate for integration into the system. Students are also required to write and maintain system-level User Acceptance Tests (UAT) based on user/customer requirements for functional and quality testing purposes. Although this testing strategy may not be defined exactly as it is in XP, it is very similar to XP's TDD, and it has evidently served the students well in familiarizing them with the different levels of software testing.

Pair programming: Each project team in the Summer 2013 class consisted of a student pair. Teams were directed to work as a driver (coder)-navigator (reviewer) pair on a single workstation, and swap roles frequently. Students were also advised to swap developer-tester roles every time they took the driver's seat so that both team members receive implementation as well as testing exposure. Virtual pair programming was encouraged when physical team meetings were not possible. Some teams, especially students who have tried a pair programming model in their industry co-op or internship assignment, suggested that this work model was not efficient for academic projects and that it is sometimes difficult to follow, especially while writing black box tests. This is contrary to findings from other case studies in undergraduate classrooms[7]. However, there was at least one team that used this model by projecting their work on a large screen and following a driver-navigator scheme during almost every programming session. This team claimed that the model helped them in both learning and productivity. Since this may not apply to all teams and/or student personas, systematic code reviews were strictly enforced and documented during every release, and the emphasis on "single workstation" pair programming was relaxed.

System metaphor: Using a system metaphor is perhaps one of the most questionable and not clearly understood practices of XP. It is essentially a "story" (a word, phrase or sentence) that describes the system's core functionality using a simple metaphor. To test the usefulness of this practice, every team came up with a system metaphor for their application during the zero feature release and used it to explain their project to on-site customers, visitors, and reviewers. It was fairly easy to find system metaphors for some applications. For example, "punch card" was used as a system metaphor to describe a mobile application that will be used by freelancers to manage billable hours and tasks for various clients. Based on the system metaphor, different components of the system (classes) were also named metaphorically, for example, clients, services, and timestamp. On the other hand, it was relatively difficult to come up with metaphors for some applications. For example, one of the teams had difficulty in finding a metaphor for an application that will generate and manage internet memes. While there are ways to overcome difficulties in finding a good metaphor[8], it does not seem particularly useful from an educational perspective.

Collective code ownership, continuous integration and refactoring: Each team maintained a central repository to version control their source code and related documentation. Teams were advised to run tests and check-in frequently so as to maintain quality during integration. While students learned the fundamentals of software maintenance, the concept of "collective code ownership" does not fit well with the idea of "independent testing", especially when there is only a semester to learn, set up and use the testing protocol. Nonetheless, when a fully-operational test suite is ready, which is usually towards the end of the release cycle (see Table II), students are exposed to practices such as refactoring and continuous integration.

40-hour week: It is assumed that students are enrolled in at least four courses during an academic semester. In order to account for the workload in all the courses, a 10-hour week (4 in-class/lab hours + 6 out-of-class hours per week) was recommended.

Coding standards: This is one of the most useful XP practices for students to learn and conform to. Hence, reference documentation and Android/Java coding standards were heavily emphasized throughout the release cycle.

On-site customer: The course instructor played the role of an on-site customer for the most part, and provided feedback after every release. Additionally, software developers and project managers from the university's IT services were invited to serve as reviewers during beta releases. This allows students to develop interpersonal skills, specifically personal effectiveness and customer interaction skills.

In summation, agile programming models such as XP or scrum, especially as presented in 14 weeks in a classroom setting, where many of its requirements cannot be adequately enforced, may not be appropriate for systems where safety and security are of paramount importance. For example, in the embedded systems industry, there is an increased emphasis on formal methods for designing and testing safety-critical systems such as medical devices. XP practices such as "user stories" would not be enough to drive specifications and testing in such applications, and much more detailed requirements would need to be elicited. Hence, computer engineers and scientists need to learn systematic design approaches such as UML, which will enable them to design reliable and secure software systems. On the other hand, it is reasonable to expose students to agile models, as there are situations where agile programming by itself is appropriate, and many engineering teams use this kind of model in industry. Therefore, some aspects of XP can be presented and integrated into the hybrid methodology.

Course Assessment

Student demographics: Ten students (5 computer science, 4 computer engineering, and 1 computer engineering technology) enrolled and successfully completed the course during

Summer 2013. The enrollment was significantly lower than the average enrollment due to the fact that this was the first summer term in the semester system and many students were on a co-op rotation in order to account for the academic calendar conversion. The small class size, however, provided an opportunity to implement and carefully assess changes in course delivery.

Course evaluation results (Summer 2013): Table III shows student responses to course-specific questions and excerpts from student feedback. While most of the excerpts come from class surveys, some comments were taken from students' reflective writing assignments. *UnLecture* evaluation results are presented in a related paper[4].Table IV shows a longitudinal assessment of student learning in this course. It can be observed that there has been significant improvement in confidence levels in the following areas: mobile programming, design and modeling, and software testing.

Table III Course Evaluation Results

Course-specific Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Using a SDLC model in the course project aided in both writing better software and understanding various software engineering principles	0%	0%	0%	10%	90%
Applying Waterfall and XP practices to the course project improved my understanding of the two SDLC models itself	0%	0%	0%	20%	80%

Excerpts from student feedback	
1	"Requirements analysis, design, and modeling in this course surpass any previous experience that I have had….UML diagrams were completely new to me, and opened up a more appealing structured design and analysis process."
2	"…learned that meticulous documentation of requirements is important as it is hard to work in teams when the requirements largely exist in only one person's head."
3	"Learning how to do a wide of array of testing was useful"
4	"I would've liked to have seen more lab classes in which we were taught about testing."
5	"…while that (design) takes up a lot of time, it does stay relatively organized. There is also less waiting around time using this (hybrid) model. You make test cases before you start, implement one goal at a time, and then test if it works."
6	"This class has given me the vocabulary to make sure I am asking the right questions. Earlier (in my co-op), I simply didn't know the questions to ask."
7	"I have enjoyed the course immensely, great content, interesting lectures, and interesting unlectures. Thank you."

Table IV Assessment of Student Learning

Expertise Assessment		Rating*		
		Pre-class	Mid-term	Final
Rate your expertise in high-level programming in general		6.8	7.1	7.6
Rate your expertise in Mobile programming (Java/Android)		3.6	5.7	7.0
Rate your "confidence-level" (or awareness) in each of the following topics	Project Management	N/A	7.1	7.9
	Requirements Analysis and Specifications		7.9	8.3
	Design and Modeling		6.4	7.7
	Object-Oriented Programming		7.8	8.6
	Software Testing		5.7	7.5
	Code Maintenance		7.1	7.9
	Ethics in Software Engineering		7.7	8.0

* Average of students ratings on a scale of 1-10, 1 being least confident and 10 being highest level of confidence

Overall, the course was successful in implementing a hybrid software development life cycle (SDLC) model for the laboratory project. “Structured design up front” and “Test-first development” are highlights of our laboratory’s hybrid model, and they are also important student learning outcomes (SLOs) of the course. Other educators have also observed similar hybrid models to be effective for classroom instruction purposes[9].

Conclusion

In summary, the software engineering course design provides simple solutions to effectively integrate a hybrid design methodology, mobile application development, and active-learning techniques. It is anticipated that this work will be especially useful for first time course developers and/or instructors interested in migrating from general-purpose/web application based software engineering courses to mobile application-based courses. Furthermore, the paper also addresses the following aspects from a classroom instruction perspective: (1) the importance of structured design and requirements analysis in building secure and reliable software systems, (2) the benefits and pitfalls of using XP in a classroom setting, and (3) the need to introduce concepts important for secure and safety-critical systems into introductory software engineering courses.

References

1. M. Shaw, “Prospects for an engineering discipline of software.” *Software, IEEE* 7, no. 6, pp. 15-24, 1990.
2. B. Boehm, “A view of 20th and 21st century software engineering.” In *Proceedings of the 28th International Conference on Software Engineering*, pp. 12-29. ACM, May 2006.
3. “SWEBOK: Guide to the software engineering Body of Knowledge” *IEEE Computer Society*, 2004.
4. V. Subbian, C. Purdy, “UnLecture: Bridging the gap between computing education and software engineering practice,” in *ASEE Annual Conference*, Indianapolis, IN, 2014 (accepted).
5. K. Beck, “Embracing change with extreme programming,” *Computer*, vol. 32, pp. 70-77, 1999.
6. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Professional, 2004.
7. K. M. Slaten, M. Droujkova, S. B. Berenson, L. Williams and L. Layman, “Undergraduate student perceptions of pair programming and agile software methodologies: Verifying a model of social interaction,” in *Proc. Agile Conf.*, pp. 323-330, 2005.

8. R. Khaled, P. Barr, J. Noble and R. Biddle, "System metaphor in "extreme programming": A semiotic approach," in *7th Int'l Workshop Organ. Semiotics*, 2004.

9. A. Shukla and L. Williams, "Adapting extreme programming for a core software engineering course," in *Proc. 15th Conf. Software Eng. Edu. and Training*, pp. 184-191, 2002.