A PRELIMINARY REPORT ON ADAPTING SOFTWARE DEVELOPMENT INDUSTRY BEST PRACTICES FOR UNDERGRADUATE CLASSROOM USE

Rajendran Swamidurai, David Umphress
Alabama State University/Auburn University
Department of Mathematics and Computer Science/Department of Computer Science and Software Engineering
rswamidurai@alasu.edu / umphrda@auburn.edu

Abstract

This paper presents an experience in designing, implementing, and evaluating an undergraduate course which incorporates software engineering industry best practices. Equipping students with knowledge about practical software development is problematic in today's academic arena. Colleges teach students the principles of software development, but that instruction is mostly theoretical and abstract. Developing working software requires specific knowledge in software engineering industrial practices. The traditional university curricula do not address these areas in any depth. We have taken a first step in departing from the traditional curricula by orienting an undergraduate course to software engineering practices. Course material on software engineering, including software process, is readily available. What is missing is the mechanism to expose students to real-world software issues encountered in the software industry. Using the four aspects of engineering as a starting point, we cataloged a number of common industry practices for accomplishing analysis, design, construction, and test. In this paper, we describe the design and implementation of the software engineering course, and then present the preliminary results and observations from the course.

1. Introduction

The courses in the computer science track expose students to a wide variety of programming languages, but that is only a small part of what is required of graduates when they enter the workforce. Industry demands a much broader perspective: that of being equipped with technical skills in identifying requirements, designing a suitable solution, implementing the solution in software, and validating that the software satisfies requirements. Industry dictates that graduates are equipped with business skills such as estimating costs, monitoring progress, measuring effectiveness, etc. Students who are inculcated with such software engineering skills are more attractive to employers than just having software-coding abilities.

U.S. Bureau of Labor Statistics, Occupational Outlook Handbook, 2010-11 Edition [1] highlights our claim. This report indicates that: 1) Computer software engineering is among the occupations projected to grow the fastest and add the most new jobs over the 2008-18 decade, resulting in excellent job prospects and 2) Employment of computer programmers is expected to decline by 3 percent through 2018. This indicates that industry is looking beyond just coding skills.

Equipping students with knowledge about practical software development is problematic in today’s academic arena. Colleges teach students the principles of software development, but that instruction is mostly theoretical and abstract. Developing working software requires specific knowledge in software engineering industrial practices. Traditional university computer science curricula do not address these areas in any depth. We have taken a first step in departing from the traditional curricula by orienting an undergraduate course to software engineering practices. Course material on software engineering, including software process, is readily available. *What is missing is the mechanism to expose students to real-world software issues encountered in the software industry.*

Examination of universities with course catalogs posted on the web, conference proceedings, and research digests shows a flurry of research on software development for software process at the graduate level, but very little evidence of software engineering practices are integrated into the undergraduate curriculum. The majority of activity in the undergraduate software process area comes from software application development in isolated senior design projects. Although use of software process in senior design is widespread, we found most universities adapting a single software process such as Extreme Programming [2]. These traditional software processes have drawbacks: 1) they focus on activities at team level but, today a significant amount of software is often written by a single person, for example, the rise of Android Market and Apple App store, where most of the apps are single-person efforts; and 2) they are tedious to practice due to their heavy-weight nature; that is, they have many rules and practice to follow. From the available software processes the Personal Software Process (PSP) [3] is the only “recognized” or published process that is tailored for one person teams. PSP is heavy-weight and encumbering due to its requirements for data collection; it doesn’t take advantage of the agile movement. Therefore, it is clear that we need a light-weight single person software process to expose the students into the real-world software engineering industrial practices.

Using the four aspects of engineering as a starting point, we cataloged a number of common industry practices for accomplishing analysis, design, construction, and test. We reviewed rated each practice in terms of 1) time to learn, 2) perceived value, and 3) tool support. Our objective in doing this was to identify those practices requiring the least amount of tool support that gave the most value for the smallest amount of instruction time. Put in the vernacular, we were trying to give the students “the biggest bang for their buck.” To that end, we found that tools used by the agile community seemed to be the best starting place. We selected the following practices: 1) Analysis using Event-Response Scenarios [4], 2) Design using CRC (Class Responsibility Collaboration) Cards [5], 3) Construction using Test-Driven Development [6], 4) Project monitoring using Time Tracking and 5) Planning using Planned Velocity/Earned Velocity.

2. Software Industry Best Practices

2.1. Analysis

The analysis phase focuses the identification of user needs. It can be thought of as envisioning a future state of the world once the software solution is in place. The analysis process includes elicitation, analysis, specification, and validation. It considers functional and non-functional specs, and seeks to be a lightweight as possible.

We used event-response scenarios [4] for analysis. Event-response scenarios treat the system under consideration as a black box that produces a reaction to a given stimulus. An event-response, or set of event-response tuples, is associated with a functional requirement. Each set becomes, in effect, an acceptance test. The process of writing event-responses starts by identifying expected behavior (the "happy path"), then by exploring software functionality under abnormal conditions (the "sad path"). The primary objectives of analysis are to specify behavior and to define acceptance tests.

#	Type	Actor	Description	Example
1	Event	Test Driver	call average with valid list	Statistics.average([1,2,3,4,5])
2	Response	Blackbox	returns average of list	3.0
3	Event	Test Driver	call median with valid list containing odd number of values	Statistics.median([1,2,3])
4	Response	Blackbox	returns median of list	2.0
5	Event	Test Driver	call median with valid list containing even number of values	Statistics.median([1,2,3,4])
6	Response	Blackbox	returns median of list	3.5
7	Event	Test Driver	call stdev with valid list	Statistics.stdev(list)
8	Response	Blackbox	returns standard deviation of list	1.29

Fig. 1: A Component Scenario

2.2. Design

We used CRC (Class Responsibility Collaboration) cards [5] for the design phase. High-level design is expressed through cards that describe the component under consideration, its functional responsibilities, and other components with which it collaborates. This technique, used heavily in the Extreme Programming community, allows students to sketch out major components with a minimum of documentation.

Component Name	
Design Approach	☐ Object-oriented ☐ Functional
Superclass	
Component Type	☐ Logic ☐ Calculation ☐ Data ☐ I/O
Collaborators	
Operations	

Fig. 2: Component Description using CRC Card

2.3. Construction

The construction phase focuses on developing a low-level design and constructing code. A red-light/green-light approach to developing code was identified as fitting the teaching criteria we laid out. In this approach, students use acceptance tests (and later, unit tests) to guide their development. For each test, the students first write a failing test case. They then write code that makes the test case pass. The code is tidied up, and the cycle begins again until all test are passed. The major advantage of this approach is to have students write tests before they begin writing code that carries out desired functionality. Provided the test cases are strong enough, passing all test cases means the software meets requirements.

Our recommended approach for software construction is,

- Construct code using Test-Driven Design,
    - write failing test case to reveal design element
    - write production code that causes the test case to pass
    - clean up the code as quickly as feasible

The objectives are to develop verifying/validating test cases and to develop the code that passes tests.

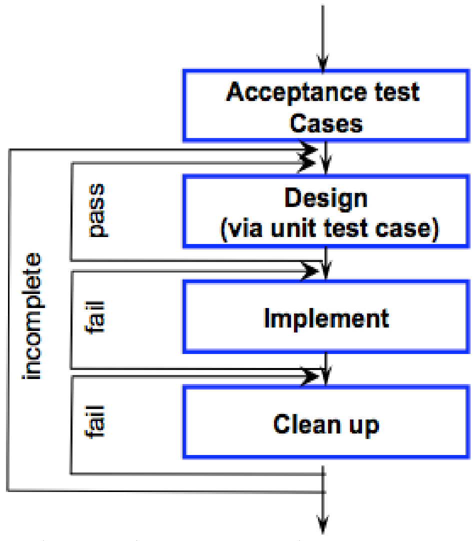

Fig. 3: The Construction Process

2.4. Project monitoring

In order for students to know how much they are putting into their development efforts, we asked that they log the amount of time, in minutes, they spent on planning, analyzing, designing, and constructing code.

2.5. Project Planning

The project planning phase focuses on estimating overall project effort given available information. *Velocity* [7] is used by the agile community to connote the amount of work completed. We adapted from the project planning community the concept of planned and earned value [8] to planned and earned velocity. Planned velocity is the number of methods that are to be written during a particular period of time and earned velocity is the number of methods actually written. Using this simple metric while developing software in successively detailed iterations allows the students to forecast new development efforts based on past performance, and gives them an insight into how well they are adhering to that forecast.

3. Implementing Software Industrial Practices in Undergraduate Course

We developed Alabama State University course CSC 437 Software Engineering II, using Auburn University's software process course, as a basis. The AU course had been taught twice a year for the past ten years, so we had quantitative data on how computer science and software engineering students performed using various industry practices. This data pointed us to those practices that met the selection criteria noted earlier. It also indicated the sequence of topics that would allow students to experience the benefit of working in a disciplined fashion. In particular, we were interested in students seeing the fruits of their studies in terms of software quality, amount of help required, and accuracy of schedule projections.

We integrated the industry practices discussed above into instructional modules each lasting one week. The modules were taught using examples that were worked through interactively during class. The students then worked on a programming assignment that incorporated the new instructional concept into concepts previously taught. This allowed the instructors to evaluate the students on their performance and to give points as to how they might improve.

The course was taught for the first time to 10 ASU students during the Spring 2013 semester. Each of the students had taken a beginning software engineering course in which they were introduced to the concepts of analysis, design, code construction, and testing. The course met for two days per week, with the first day being devoted to presenting material and second being used as a supervised lab period. The students implemented their software in Python using the Eclipse development environment and the Pydev Eclipse plugin. Scenarios, time logs, defect logs, CRC cards, acceptance test reports, and schedules were recorded in Excel spreadsheets.

The topics presented, in order, were

- Week 1: Software engineering overview
- Week 2: Lifecycles
- Week 3: Python, Eclipse, unit test
- Week 4-5: Construction using TDD
- Week 6: Informal reviews
- Week 7-8: Analysis using scenarios
- Week 9-10: Design using CRC cards
- Week 11-12: Estimation using velocity
- Week 13-14: Scheduling using calendars
- Week 15: Wrap up and feedback

4. Summary and Conclusions

We adapted the software industry best practices for a typical computer science undergraduate software engineering course and evaluated its effectiveness through student feedbacks. We feel the course was a moderate success, but indicated there was room for improvement. Several relevant observations emerged from a feedback sessions and anonymous course comments: 1) Students came to the course with weak development skills. This necessitated an unexpected amount of time having to be devoted to using Eclipse and writing rudimentary Python. We had to allow extra time for each assignment, and dropped the last two programming assignments because of lack of time. 2) Students expressed surprise at having their software tested by the instructors to the level of boundary value analysis coverage. They noted that the amount of work required to follow a disciplined approach was perceived as being daunting. 3) Students did not have an industrial context in which to judge the merit of the practices were asking them to use. Several of the students had been on industry coops, but none had been exposed to lifecycle activities other than writing code. Asked if they would continue to use the techniques taught in the course, the majority answered "no," with "too much effort" cited as the reason. A formal evaluation of the course is underway; we are quite optimistic that the practice centered learning model will prove to be an effective approach to reinvigorating software engineering education.

5. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1238912.

6. References

1. *U.S. Bureau of Labor Statistics, Occupational Outlook Handbook*, 2010-11 Edition http://www.bls.gov/oco/ocos303.htm#projections_data.
2. Kent Beck, *Extreme Programming Explained: An Embrace Change*, Addison-Wesley, 2000, ISBN 0201616416.
3. Watts S. Humphrey, *PSP(sm): A Self-Improvement Process for Software Engineers*, Addison-Wesley Professional, March 2005.
4. Stephen Montgomery, *Object-Oriented Information Engineering: Analysis, Design, and Implementation*, Academic Press
5. Beck, K. & Cunningham, W., *A Laboratory for Teaching Object-Oriented Thinking*, OOPSLA'89 Conference Proceedings (1989), pp. 1-6.
6. Beck, K. *Test-Driven Development by Example*, Addison Wesley - Vaseem, 2003
7. Velocity, http://guide.agilealliance.org/guide/velocity.html
8. Humphreys, Gary, *Project Management Using Earned Value*, Humphreys and Associates, 2001, ISBN 0-9708614-0-0