

Foundational Actions: Teaching Software Engineering When Time Is Tight

Jerry Boetje
Computer Science Department
College of Charleston
Charleston, SC 29424
jboetje@cs.cofc.edu

ABSTRACT

Often Software Engineering courses approach educating undergraduates in good processes and practices by using a simulated product development environment, following all of the steps for product development in a single semester. Some also create multi-semester projects to improve the student experience. We are in the group using multi-semester projects, but our approach differs in that we have only a single semester of project work per team, focused on the core actions used in all processes from waterfalls to XP. We call this foundational actions. Using industrial tools, processes, and evaluation methods, the students develop and integrate components of a well-specified, but major product. Emphasis is placed on teamwork, communication, and ultimately, working production code created with the foundational actions. The approach has also spun off related independent study opportunities for advanced students and even non-CS majors.

Category and Subject Descriptors

D.2.9 [Management]: *Life Cycle, Programming Teams, Software configuration management*

General Terms

Documentation, Legal Aspects, Design, Experimentation

Keywords

Education, Pedagogy, Software Engineering, engineering, process, software, tools, undergraduate,

INTRODUCTION

From the vantage point of a 30-year career in software development and management, the author participated in the development of what we now call Software Engineering, starting at Draper Labs with the Apollo project, through DEC and the minicomputer era, to the frenetic creativity of Silicon Valley. Fads came and

went, each one leaving its mark on the process of building and delivering software. Over the the past 5 - 8 years, the industry has begun to settle on a few sets of “best practices”, e.g. CMM [5], XP[1], and spiral [6, 7]. Each set addresses meta-needs and trade-offs of a project such as extreme reliability, adaption to new requirements, and flexible integration with other systems. Each of these practices has collected supporting toolsets appropriate to the practice. While this development is good for the industry, it places a larger burden on educators faced with teaching such a broad range of practices. This burden falls more heavily on the CS departments of schools such as the College of Charleston where a broader liberal arts curriculum cannot support a 3 or 4 semester sequence [4] just for Software Engineering (SE).

Given our curriculum constraints, we adopted a two-semester sequence consisting of a SE theory and book course required for junior or senior students followed with a practicum course mostly for graduating seniors where they apply the learning from the previous course to a real problem. This is not a novel solution, nor is our approach of building a large-scale, multi-semester system[4, 8]. But our focus on teaching foundational actions rather than specific processes appears to be so.

1. FOUNDATIONAL ACTIONS

In a single semester, it is just not possible to both give students a complete experience of a significant process and avoid the predictable problems evinced in [4]. And attempting to use multiple processes simultaneously would be difficult and confusing. At CofC we adopted a foundational action (FA) approach. FA focuses exclusively on the proper use and integration of the practice fragments (design, architecture, code, test, documentation) and the common set of tools found in all of the major processes: source code control, integrated development environment, document repository, status reporting, defect tracking, system and unit test harness, automated build system, and discussion forum. While there is always a clear process in each semester, it seldom conforms strictly to any of the common processes. But it always contains these elements.

2.1. Proposition

Our approach to foundational actions in software engineering education rests on the proposition that there are a constant set of features and actions that run through all successful SE processes. It is these features and actions that must be taught and practiced regardless of the specific overall process. By concentrating on these components embedded in an appropriate process in each

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITiCSE'06, June 26–28, 2006, Bologna, Italy.

Copyright 2006 ACM 1-59593-055-8/06/0006...\$5.00.

semester, the students will have both the knowledge and the flexibility to integrate well into an industrial setting.

2.2. Description

The foundational action approach dispenses with some components of a major process. In particular, we started the project with a well-defined specification of the product [2, A1], thereby removing the requirements definition phase. Interestingly, this does not remove the cycle of understanding present in all processes. All complex systems contain ambiguities in their specifications, discovered in the design/implementation phases of each semester, and requiring some research or resort to the project architect (the instructor).

In dropping the client interaction process component from the overall requirements definition, we are open to some criticism. However, the author (with many years experience in this area) contends that most undergraduates are not ready to do that job without significant experience and some amount of mentoring. Our trade-off is to allot that time to the other aspects of the job they will be expected to perform.

Each semester students design, implement, test, and integrate one or more components of the overall product. As they work, they must follow a basic development process, report status, participate in online discussions on issues, fix bugs, and update documentation. The tools they use are ones commonly used in major industrial development efforts, implying they are big and complex. All are Open Source tools excepting the source code control system which is more flexible and powerful than the current Open Source tools (CVS and Subversion).

A semester's process may differ from prior ones based on the nature of the product components, the size of the team (varies from as few as 6 to as many as 25), and the personalities of the team.¹ But the foundational actions are always present, providing a commonality of experience for students that they take to their careers.

2.3. Student Experience

We saw in this approach an opportunity to give our students real-world experiences not usually available in an academic setting but ones they will encounter in industry no matter the specific process in use:

- **Sense of overwhelm.** Being dropped into a large and very complex environment where they will sink before they start swimming.
- **Getting control.** Learning and using complex tools to cope with complex development requirements.
- **Sophisticated process.** The need for software engineering is not so apparent in small, semester-long projects.
- **Complexity.** Many undergraduates have never build an application that exceeds 50 classes or 2000 lines of code (SLOC).

¹ This is an aspect of teams that is often neglected in the choice of appropriate processes.

Being about 1/2 finished, our product is now over 700 classes, 50,000 SLOC, in 2 languages.

- **Customer/requirements focus.** Having to remember and balance many requirements and concomitant tradeoffs in design and development.
- **Experiential learning.** Doing it rather than reading or hearing about it.
- **Personal judgement.** Ability to assess the development process used (or not!) by a potential employer.

2.4. Core Invariants

These are the constants we use to anchor the course.

- **Product.** One that could not be completed in one or two semesters, but could be broken down into semester-sized chunks, involves a range of architectural and programming techniques, has an existing specification, and can be integrated with Open Source or public domain code as appropriate. The decision came down to the background and experience of the primary instructor since he would of necessity be the overall architect.
- **Tools.** Industrial-strength and commonly used tools, available to the college at no cost, do not require an inordinate amount of IT support, and not so complex that they are unusable in the time frame. Students will learn and work with many different development environments in their careers. Our environment is deliberately non-uniform allowing us to change as needed rather than be locked into a single-vendor solution. And some such as RUP[3] are too complex for the level of our project.
- **Processes.** The overall product development process is a spiral with each semester being one turn. Each turn encompasses all of the four phases of the mini-development cycle: inception, elaboration (including throw-away prototypes), construction, and transition. .
- **Licensing.** Consistent with our academic role, it was decided to control the distribution and use via an Open Source license.² This exposes the team to licensing issues, particularly as we integrate other Open Source code.

3. ACADEMIC REQUIREMENTS

3.1. Grading Policy

The goal is to judge each student's performance in a manner similar to the evaluation methods used in industry. The final grade represents a blend of delivery of correct code, attendance, teamwork, timeliness, communication, and initiative. The criteria are:

- **A** - Same as for B, but the student must demonstrate significant initiative. Initiative is defined as an action a student takes that improves the design or implementation and that was not as-

² It is currently licensed under the GPL2. We will be moving to the MIT license at the request of a potential user.

signed as a task. For example, a student realizes there are two possible implementations and runs a well-designed and documented benchmark of the possibilities.

- **B** - Good attendance, teamwork, communication, and generally on-time delivery.
- **C** - Poor execution on two or more of the parameters for B.
- **F** - Little or no execution on assigned tasks is grounds for “firing”. It happens.

3.2. General Semester Structure

Each semester has a basic (but not rigid) structure:

1. **Inception** (aka the “sink”). The period when the new team gets its bearings. They learn about the tools, the development process, the architecture of the product, and what they’re expected to create during that semester. To practice using the tools in the project environment they also fix a few simple bugs left over from a previous semester and update the documentation. This phase typically lasts 3 weeks. In the current semester, the team cut the inception period to 1 week. We attribute this acceleration to the maturity of the process and the supporting engineering documentation which is augmented and edited by each team.
2. **Elaboration**. The period when the team works out how to accomplish the goals for the semester. This involves discussing and documenting their designs as well as writing some code to try out their ideas. This takes 2 to 3 weeks depending on the difficulty of the semester goals: a given semester may have one complex or multiple simple goals.
3. **Implementation - code**. The period when they write both the product code and the test code. Writing test cases during coding is an essential aspect of the process. This is typically 3 to 4 weeks.
4. **Implementation - integration**. The period when all of the new code is merged into the product. This is sometimes simple, such as adding new types. Other times it’s a larger process requiring changes in many existing functions and classes. With the time remaining, the students fix more bugs and continue to update documentation.

The course places a premium on succinct and well-structured documentation useful for succeeding teams to understand not only *what* was done but *how* and *why*. This documentation suite encompasses the official design specifications, the Javadoc API documentation, and the discussion forum. One of the tasks of an incoming team is to update project documentation as they come up to speed. For example, the HowTo topic improves every semester.

5. **Transition**. The nominal final exam period is used for student presentations of their work that semester. Every student must

present with the instructor being the critical audience. These presentations are be recorded and available online to future teams. Note that the documentation and presentations also provide a valuable transition function to the incoming team - an activity often missing in semester-long projects.

The actual structure for a semester mixes these various phases as appropriate. For example, the first semester consisted of a building and integrating a few, well-defined core components, leading to use of a modified waterfall process. The current semester (the sixth) consists of several small, complex components that are not easily separated for development leading us to a form of agile development process.

4. BENEFITS OF THIS APPROACH

4.1. Mainstream Students

This class is designed specifically for them. It gives them an experience of the industrial environment in a “safe” setting. It is also hard to fail if they put in the work. Some courses that attempt to handle the breadth of SE process run the risk a failing project from poorly defined requirements. While this certainly happens in the real world, the author contends that students will see enough of that in their career. At this point, a success is more useful.

4.2. Advanced Students

This project also opens opportunities for more advanced students to take on very sophisticated and challenging tasks. So far we have had one Bachelor’s essay completed (the bootstrap compiler), another in process (automated XML-based documentation), and a Master’s thesis (mapping of the CLOS meta-object protocol onto the Java object model).

4.3. Other Opportunities

It’s no surprise that student capabilities and desires differ significantly. The size and complexity of our project creates many niche roles beyond the classic developer. Students have already taken on the following roles:

- **Quality Assurance (SQA)**. Creation of white and black box tests for product components.
- **Web designer**. A Communications major with a CS minor learned how to organize information about the project and the product, and design and implement a web site.
- **Tool maintenance**. Defining, implementing and enhancing components of the development process.
- **Team lead**. Coordinating a small team to handle a specific goal.

Other roles that may eventually branch from development are:

- **Business-related**. There are opportunities for business students to be involved in roles such as product manager, market research, and communications.

- **Documentation.** Developers are sometimes not very good at written communications. Students interested in technical writing create components of the product documentation.

4.4. Surprises

Since we anticipated that our approach would have difficulties, there were few bad surprises. One problem was a group that never developed a sense of a team and mutual support. It was a small group, and the specific personalities never meshed. The effect of this non-team was poor designs and code that later groups had to repair. Of course, we could argue that it was an excellent learning opportunity for later teams. This was a clear anomaly since all the other teams have meshed well by the mid-semester.

We initially supported two blogs, one for status and announcements and a second for general discussion of topical issues. The first works well for the team and helps structure the week with review of accomplishments and goals for the upcoming week. The second was a flop. The students preferred to talk among themselves to solve problems or finish designs. No amount of prodding seemed to help. This meant that we lost the record of these useful discussions. In Fall 2005, we switched to a web-based forum. The students are prolific on the forum, carrying on detailed discussions that are recorded for later review. We believe that the topic-centric nature of a forum made it more a help than a management requirement.

5. CONCLUSIONS

After several semesters, we have achieved the goal of creating an industry-like environment that is not tied to a specific SE process. The process of breaking the development into semester-long chunks works, albeit with a bit of scramble at the end of the semester - just as in industry before ship. We continue to tune the teaching process along with the development environment. In particular, the transition phase at the beginning and end of the semester gets smoother and more effective. The process has created un-looked-for opportunities for advanced students and even non-majors.

A. APPENDICES

A.1. Project Description

CLforJava[2] is an original implementation of Common Lisp (CL), a large, complex language that includes a large library of tools. CLforJava differs from other CLs in its insistence on intertwining with Java. The breadth of the implementation supplies a myriad of tasks ranging from simple coding to Masters' Thesis.

A.2 Tool Suite

Document Repository	TWiki (twiki.org)
Source Code Control	Perforce (perforce.com)
Java IDE	Netbeans (netbeans.org)
Discussion forum	SMF (simplemachines.org)
Status Reporting	Moveable Type (sixapart.com)
Defect Tracking	Bugzilla (bugzilla.org)

6. REFERENCES

- (1) K. Beck, "Planning Extreme Programming", Addison-Wesley Professional, 2001
- (2) J. Boetje, "Common Lisp for Java: A New Implementation Intertwined with Java", Proceedings of the International Lisp Conference, 2005
- (3) P. Krutchten, "The Rational Unified Process: An Introduction", Addison-Wesley Professional, 2004
- (4) M. Sebern, "The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment", Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02), IEEE, 2002
- (5) Software Engineering Institute, "Capability Maturity Model: Guidelines for Improving the Software Process", Addison-Wesley, 1995
- (6) I. Sommerville, "Software Engineering, Seventh Edition", Addison-Wesley Professional, 2005
- (7) F. Tsui, "Managing Software Projects", Jones and Bartlett Publishers, 2004
- (8) N. Wilde, et al, "Some Experience With Evolution and Process-Focused Projects", Proceedings of the 16th conference on software Engineering Education and Training, IEEE, 2003