Removing the Emphasis on Coding in a Course on Software Engineering

Linda Rising[1]
Department of Computer Science
Indiana University-Purdue University at Ft. Wayne
Ft. Wayne, IN 46805
and
Magnavox Electronic Systems Company
Ft. Wayne, IN 46808

ABSTRACT

There has been considerable interest in a one-semester course in software engineering [Bullard88, Carver87, Gibbs87]. Faculty members of departments of computer science are introducing courses that involve team projects, in an effort to provide students some experience with large programs . However, software professionals are still concerned that most computer science graduates have little understanding of what is involved in the development of large, complex systems. Too often, code alone is regarded as the primary product without proper consideration of the necessary standards and procedures of the controlling disciplines. This paper describes a course that shifted the emphasis from coding by having students perform supporting activities and maintenance on a large Ada project.

INTRODUCTION

In addition to serving as assistant professor in the Department of Computer Science at Indiana-Purdue at Ft. Wayne (IPFW), the author also works as a consultant for Magnavox Electronic Systems Company, a company that has been involved for the last several years in the development of a large Ada project. The opportunity to see some of the problems encountered has resulted in an increased emphasis on software engineering principles in all the author's courses. In addition, a senior-level course, CIS 474 Topics in Software Engineering, was introduced in Spring 1988.

IPFW offers a large project course for sophomores. The students work in groups of three or four and have a chance to see some of the problems that arise when working in a team but little or no time is spent on important topics such as cost estimation, formal requirements, quality assurance, configuration management, and technical reviews. This course is like many reported in the literature; each student is involved in the requirements, design, coding and testing phases. Since it must be completed by the end of the semester, the project cannot be too large and the focus ultimately becomes coding.

Although it does occupy 95% of a student's programming time, coding accounts for less than 25% of the effort for a large government project with significant document requirements, independent verification and validation, and other ancillary tasks [Jones86]. In addition, it becomes painfully obvious to anyone who writes long-lived software that correctness is a moving target. Boehm points out [Boehm81] that many of the characteristics of good software are in conflict with each other and that trying to achieve them all is impossible. Programmers are smart people who will work hard to achieve desired goals. Trying to write programs that are correct usually means sacrificing modifiability, readability, maintainability, etc.

It is important that students understand that all those "abilities" are more important than correctness alone, since most software must be modified, read, etc., to produce code that is, at best, only temporarily correct. Unfortunately, courses in computer science emphasize, by default, code and correctness. An instructor does not have time to reinforce notions of good design, style, and documentation when grading programming projects. As a result, student programs are graded only on correctness, run against a bank of test cases. What students learn from this is that correct code is the only important component of good software. Software engineering ideas presented in class should be reinforced when student work is graded. At IPFW, an attempt has been made to solve this problem by developing department standards for program grading that are enforced in all classes [Rising87].

The interest in software engineering has come about because of what has come to be called the "software crisis." Typically behind schedule and over budget, many software projects are so inadequate that they are

[1] The author is currently a graduate student at Arizona State University.

never used. As reported by DeMarco [Demarco82]:

> "15% of all software projects never deliver anything; that is, they fail utterly to achieve their established goals.
>
> Overruns of one hundred to two hundred percent are common in software projects."

This is costly and wasteful of resources. It is imperative that better methods be used.

One response to the software crisis, on the part of the Department of Defense, has been the founding of the Software Engineering Institute (SEI) to "bring the ablest professional minds and the most effective technology to bear on rapid improvement of the quality of operational software in mission-critical computer systems." One of the activities of the SEI is the sponsoring of Faculty Development Workshops where curriculum modules are presented. A curriculum module presents a topic in software engineering and consists of an outline, brief descriptions of important component areas within the topic, an annotated bibliography, and suggestions for teaching. The modules can be tailored to individual needs in a university or industrial training setting. The author has attended three of these workshops and wanted to include as many of the modules as possible into the new course.

A resource available to the author through Magnavox is the Ada Repository. It contains reusable software components and tools. Several of these were chosen for use in the course. One, an Ada style checker, had been modified by the author for use at Magnavox. The ease with which this modification had been performed determined that the correction and possible modification of this program would be the project for the course.

COURSE DESCRIPTION

The course, CIS 474 Topics in Software Engineering, had an enrollment of nine seniors who were assigned roles following the guidelines in Tomayko's [Tomayko87C] report on a one-semester course in software engineering:

Principal Architect: Bears primary responsibility for the creation of the software product. Primary responsibilities include writing the requirements document, advising on overall design, and supervising implementation and testing. Also calls and conducts change control board meetings.

Project Administrator: Responsible for resource tracking. Primary responsibilities include cost analysis, investigation and use of a manpower tool, and cost control. Develops form for weekly resource reports. Collects data and issues weekly cost/labor consumption reports and a final report. Also serves on change control board.

Configuration Manager: Responsible for change control. Primary responsibilities include writing the configuration management plan, developing forms for change requests and discrepancy reports, tracking change requests and discrepancy reports, and preparing product releases.

Quality Assurance Manager: Responsible for the overall quality of the released product. Primary responsibilities include preparing the quality assurance plan, calling and conducting reviews, and evaluating documents. Will investigate and use McCabe's Metrics tool.

Tester: Responsible for creation and execution of test plans to verify and validate the software, including tracing requirements. Will investigate and use testing tools.

Designer: Responsible for producing design documents for the product. Will investigate and use Excelerator to prepare preliminary and detailed design documents.

Implementor: Responsible for developing coding standards, implementing the changes in designated modules, and writing the user interface. Will conduct code reviews.

Document Specialist: Responsible for the development of documentation standards and the user manual. Will assist in the development of the user interface and the preparation of uniformly formatted documents.

Each student was given some reference material to read to become familiar with the role he/she was to play in the class. Those students who were responsible for preparing documents were given copies of the IEEE standards for that document [IEEE84] as well as the sample documents included in Tomayko's support materials [Tomayko87C]. The lectures were scheduled to cover topics necessary for meeting scheduled milestones. In addition, outside speakers on most topics were invited to talk about their duties.

Originally, the goal for the class had been to modify the style checker so that the style being checked was that of the department at IPFW. Midway through the semester, it became apparent that this was too ambitious. Since the author had previously modified the style checker and knew that (at least) three errors existed in the code, these errors were reported individually to provide experience in tracking, correcting and testing.

Thus, the focus of the course was truly removed from code and emphasized the controlling disciplines.

SOME OF THE TOPICS

One of the most important topics covered in the class was technical reviews. Initially this is a difficult concept for students, since, in all other classes, they are not encouraged to work together or to examine one another's work, and never spend class time involved in the activity. An introductory lecture was given on technical reviews, using the material in the SEI curriculum module [Collofello87]. Some instructors feel that they must be present to direct student reviews. The author agrees with those who say that "first-line managers" hamper the review process. Reviewers care more about what the "boss" will think than about doing a careful review. The author attended the first review to reinforce points from the lecture, but the Quality Assurance Manager directed all subsequent reviews. A report summarizing the final decision reached in each review was submitted but did not include the Action List. The students were not graded on the number of errors found or the number of times their work required review, but were graded on their preparation for the review. Peer evaluations done at the end of the semester reported whether each team member felt that the others had been active participants in the review process. Not only were students learning about the review process but since all plans, standards, and reports were reviewed, preparing for the review gave each student a chance to study all of the documents.

Another important topic was Configuration Management (CM). An SEI module, [Tomayko87A], and support material, [Tomayko87B], were used to present an introductory lecture. The CM Manager prepared a CM Plan which outlined the following process for treating errors in the program. Team members reported all problems with the style checker to the Change Control Board (CCB) by submitting a Discrepancy Report (DR) or Change Report (CR). The CCB met and approved/rejected the discrepancy/change. The implementors received a copy of the DR/CR and repaired the problem. After code review and separate compilation, the changed module was sent to the CM Manager, who reconfigured the style checker and reported the location of the new version to the tester. The use of Ada allowed individual modules to be compiled without allowing the implementors access to the rest of the system. The tester first tested the change and then did regression testing. Successful testing was reported to the CM Manager who signed the DR/CR as repaired and submitted it to the CCB. The new release was reported to the entire group using system mail. The size of the style checker (62 files, 787 blocks, approximately 15,000 lines of code) made good CM necessary. The students saw clearly how serious problems could arise in version control with a large team and lack of standards and procedures for CM.

A third important topic covered in the course was Quality Assurance (QA). Again, there was an introductory lecture using an SEI curriculum module [Brown87]. The QA Manager had as her primary responsibilities preparing the QA Plan and conducting technical reviews. The QA Manager also evaluated each module in the program initially using a McCabe's Metric tool and after each modification checked to see that the complexity of the changed module did not increase significantly. An increase would have resulted in the submission of a DR to the CCB. The complexity remained the same for all changed modules in the project.

PAPERS AND SPEAKERS

In addition to the two texts used in the class, [Fairley85] and [Brooks82], the students were required to read a collection of papers on several software engineering topics. Most of these papers reported results of industrial experiments or observa-tions. These papers are listed in the REFERENCES followed by an *. Each student chose one of these papers and presented a ten-minute summary during the last week of the course. To insure that all the students read all the papers, each presentation was evaluated in a brief paragraph submitted by each of the other students.

It was made clear to the students that these were not esoteric or academic subjects but essential and practical topics. To empha-size this, several speakers from local industry were invited to make presentations to the class. These presentations involved problems and solutions of a very practical nature. The speaker on Design introduced the somewhat overwhelming problem of dealing with government standards, especially DOD-STD 2167. The speaker on Configuration Management shared his experiences with control-ling, what had been at the time, the largest Ada project in the world. The topics covered in class seemed to become more important when someone who worked in that area reinforced what had been presented in class. The speaking dates were scheduled at the beginning of the semester and invitations were extended to anyone in the community. There were always visitors for each presentation, from the university or local industry.

LESSONS LEARNED

One significant problem during the semester, which had been anticipated but still proved a severe difficulty was the number of restrictions on a student account. The tools were very large, and size and other quotas imposed on students proved a significant handicap. Requests to the system manager could not be made by the students, so often the author would become a temporary team member, completing a compilation sequence or solving some system difficulty. It was sometimes enjoyable, sharing the students' perspective

but many times it was frustrating to have to deal with what seemed, in many cases, like arbitrary restrictions on student accounts.

Readers may have noticed that the class had nine members but that there were only eight roles. It was decided to have two implementors since no one who wanted that job knew Ada. This also enabled each to do code walkthroughs for the other. However, experience with the project leads to the conclusion that it would have been better to have had two designers. The designer was the only one to prepare two documents, and with a program of this size, that turned out to be a mammoth chore. The code walkthroughs could have been done by any of the three Ada experts in the course and, in fact, they often served as consultants for implementation problems.

The department had recently acquired copies of Excelerator, a software tool to aid in design documentation. In order to provide some experience with this tool and hopefully create easily modifiable design documents, the designer investigated and used the tool for both design documents. Unfortunately, Excelerator is best used for structured design and our designer had some difficulty using it for packages and dependencies.

This group was too large for maximum effectiveness in the technical reviews. A group of four or five is recommended. The final reviews were done in two simultaneous sessions, each with half the class. Each individual was more effective in the smaller group, since he knew his observations were not going to be made first by the stronger, more vocal members. In earlier reviews, this caused some to be intimidated to the point of not participating or to come to the reviews unprepared. Unfortunately, this wasn't discovered until the peer evaluations were read at the end of the course.

Everyone in academia and industry knows that there is a noticeable variation in ability and enthusiasm within a group of any size [Sackman68]. This is sometimes not as evident in a more structured class situation but in a course like this where the quality of the product depends a great deal on the energy expended by the individual team member, it is obvious that what Brooks calls "hustle" [Brooks82] is an important factor. This makes grading difficult. How should grades be assigned to those who do the minimal amount of what must be done and those who contribute extra effort for the project? Do they both deserve an A? The students are aware of this and in their peer evaluations they mentioned the exceptional contributions of three of the students on the team. The author's solution was to try to duplicate real world conditions and give a "bonus" to these students by relieving them of preparing the evaluation of the paper presentations.

CONCLUSION

The students agreed that they had enjoyed the course and learned a lot. Some comments made during the semester:

> "I really have learned to see The Big Picture."
>
> "I never realized how much paper is involved in all this."
>
> "Why can't they (industry personnel, managers, etc.) see that this is the way to do things right?"

The need for increased understanding of "how to do it right" has never been greater. Our class began each meeting with a student's reading of one or two accounts of software disasters from issues of ACM SIGSOFT Software Engineering Notes. It's clear that instead of dissipating, the software crisis has gotten worse. As Brooks said recently in a keynote speech at the SEI [Gibbs87]:

> "The peak year in sales for **The Mythical Man-Month** was only two years ago. Yet the book was written in 1975, about an experience in 1963-1965. The fact that it has the slightest relevance now is a sad comment on the progress of the discipline."

In the author's opinion, this class represents a step in the right direction to a solution for the software crisis. The students in this class are now software engineers, not just programmers. They understand that good software development is much, much more than temporarily correct code.

REFERENCES

[Baker72] Baker, F.T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, vol. 11, no. 1, pp. 56-73, 1972. *

[Boehm81] Boehm,B., **Software Engineering Economics**, Prentice-Hall, 1981.

[Boehm87] Boehm, B., "Industrial Software Metrics Top 10 List," IEEE Software, Sept 87, pp. 84-85.

[Brooks82] Brooks, F.P., **The Mythical Man-Month: Essays on Software Engineering**, Addison-Wesley, 1982.

[Brooks87] Brooks, F.P., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, vol. 20, no. 4, pp. 10-19, April 1987. *

[Brown87] Brown, B.B., "Assurance of Software Quality," SEI-CM-7.

[Bullard88] Bullard, C.L., et al, "Anatomy of a Software Engineering Project," ACM SIGCSE Bulletin, vol. 20, no. 1, Feb 1988, pp. 129-134.

[Carver87] Carver, D.L., "Recommendations for Software Engineering Education," ACM SIGCSE Bulletin, vol. 19, no. 1, Feb 1987, pp. 228-232.

[Collofello87] Collofello, J.S., "The Software Technical Review Process," SEI-CM-3.

[DeMarco82] DeMarco, T., **Controlling Software Projects**, Yourdon Press, 1982.

[Fagan76] Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, no. 3, pp. 182-211, July 1976. *

[Fairley85] Fairley, R., **Software Engineering Concepts**, McGraw-Hill, 1985.

[Gibbs87] Gibbs, N.E. and R. E. Fairley (editors), **Software Engineering Education: The Educational Needs of the Software Community**, Springer-Verlag, 1987.

[IEEE84] **IEEE Software Engineering Standards**, Wiley-Interscience, 1984.

[Jones86] Jones, T.C., **Programming Productivity**, McGraw-Hill, 1986.

[Lehman80] Lehman, M.M., "Programs, Life Cycles, and Laws of Software Evolution," Proc. IEEE, vol. 68, no. 9, pp. 199-215, Sept. 1980. *

[Meyers78] Meyers, G.J., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," Comm. ACM, vol. 21, no. 9, pp. 760-768, Sept. 1978. *

[Parnas85] Parnas, D.L., "Software Aspects of Strategic Defense Systems," ACM SIGSOFT Software Engineering Notes, vol. 10, no. 5, pp. 15-23, Oct. 1985. *

[Parnas86] Parnas, D.L. and P.C. Clements, "A Rational Design Process: How and Why to Fake it," IEEE Trans. Software Engineering, vol. SE-12, no. 2, pp. 251-257, Feb. 1986. *

[Rising87] Rising, L.S., "Teaching Documentation and Style in Pascal," ACM SIGCSE Bulletin, vol. 19, no. 3, pp. 8-14, Sept. 1987.

[Sackman68] Sackman, H., et al, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," Comm. ACM, vol. 11, no.1, pp. 3-11, Jan. 1968. *

[Tomayko87A] Tomayko, J.E., "Software Configuration Management," SEI-CM-4.

[Tomayko87B] Tomayko, J.E. (editor), "Support Materials for Software Configuration Management," SEI-SM-4.

[Tomayko87C] Tomayko, J.E., "Teaching a Project-Intensive Introduction to Software Engineering," CMU/SEI-87-TR-20, ESD-TR-87-171.

Curriculum modules or support materials may be ordered from the Software Engineering Institute:

SEI Education Program
ATTN: Curriculum Request
Carnegie Mellon University
Pittsburgh, PA 15213

Tape copies of the Ada Repository are available on 9 track, 1600 bpi, ANSI formatted magtapes; the charge is $200 for 3 tapes. A lower price can be negotiated if tapes are supplied.

Navajo Technology Corporation
Navajo Nation
Box 100
Leupp, AZ 86035
602-686-63791