Software Engineering Beginning In The First Computer Science Course [1]

Jane C. Prey James P. Cohoon Greg Fife

Department of Computer Science
School of Engineering and Applied Sciences
University of Virginia
Charlottesville, VA 22903

Abstract. The demand for computing and computing power is increasing at a rapid pace. With this demand, the ability to develop, enhance and maintain software is a top priority. Educating students to do competent work in software development, enhancement and maintenance has become a complex problem. Software engineering concepts are typically not introduced in beginning computer science courses. Students do not see software engineering until the third or fourth year of the curriculum. We do not believe students can acquire an adequate software engineering foundation with the present approach. We believe an emphasis on software engineering should begin in the very first course and continue throughout the curriculum. We are redesigning our curriculum to reflect this. The first course of the new curriculum is complete. This article focuses on two of the laboratory activities we have developed which deal with specific software engineering concepts.

Introduction

Computer science has changed rapidly in its brief history. What we once considered necessary subject areas, skill and knowledge levels are now inadequate. Today's computer professional must have an extensive set of skills and detailed knowledge in many different technical areas. Similarly, the individual entering the research community needs a thorough grounding in many diverse areas.

The Department of Computer Science at the University of Virginia is part of the School of Engineering and Applied Sciences. From this vantage point, we have embraced a strong commitment to the engineering aspects of computer science. We seek to educate computer scientists with a clear understanding of, an appreciation for, and skills that support the engineering and comprehension of large software systems, reengineering of existing systems, use of modern tools and environments, and application of innovative techniques such as software reuse.

Problem Statement

The demand for computing and computing power keeps on increasing. “Indeed, computing and increasingly powerful computers are the driving force behind the movement of society into the information age, affecting transportation, finance, health care, and most other aspects of modern life; computing technology and related services account for about 5% of the gross national product”. [1] With the demand for software becoming a larger and more significant portion of the GNP, the ability to develop, enhance and maintain the software is a top priority.

Educating students to do competent work in software development, enhancement and maintenance has become a complex problem. We are not presently providing students with the skills, knowledge or experience to successfully handle the new challenges found in today's software.

Traditionally, computer science curriculum emphasizes:

- the construction of relatively small programs, at most a few hundred lines;
- the use of a programming language that is rarely used outside of undergraduate courses;
- development of programs “afresh” for each assignment or course;
- a development environment lacking modern tools;

[1] The work detailed in this paper was supported in part by NSF grant CS-NSF-5239-92.

- programming in isolation or in small groups at best;
- the belief that if a program “works” it is acceptable;
- informal development approach rather than one which is rigorous and exercises analytic skills.

Software engineering concepts are not introduced in beginning computer science courses. Students do not see software engineering until the third or fourth year of the curriculum. Students take courses in algorithms or syntax and semantics of a particular programming language as prerequisites, but these courses are not true software engineering. Software engineering is left to one course which must squeeze both its theory and practice into one semester, and this theory and practice is not reinforced by use in subsequent courses.

Why Change?

We do not believe that students can acquire an adequate software engineering foundation with the present curriculum approach. “The problem is a serious mismatch between what is taught, how it is taught, and the emphasis it receives on one hand, and what the consumers of the education actually need on the other.” [2]

Comparing the content of the curriculum with the situation in the real world, we see a considerable contrast. Practicing computer scientists and software engineers deal with the antithesis of what we teach:

- software systems that are often thousands or even millions of source lines long;
- tasks that involve modifying such systems rather than developing them;
- existing systems that might be very old but remain important and have to be maintained;
- tool-rich working environments;
- development efforts that are undertaken by large teams;
- the realism of cost/performance trade-offs in business contexts;
- system development according to mandated processes and standards;
- expenditure of considerable effort on tasks other than source-code development.

We believe there must be a shift of emphasis in the computer science curriculum to meet these needs. Our new curriculum is driven by the desire to provide the education needed for competent computer and software engineering professionals. These changes necessitate a complete revision of the content, approach, and resources used in the undergraduate program.

We believe that an emphasis on software engineering must begin in the very first course and continue throughout the entire curriculum in order to provide the foundation necessary.

Approach

The Computer Science Department at the University of Virginia has undertaken a shift of emphasis in the undergraduate computer science curriculum to meet the needs outlined above.

This approach emphasizes:

- a philosophy of engineering incorporated into all of the core courses;
- an emphasis on software engineering beginning in the very first course and continuing throughout the curriculum;
- a high degree of mathematical rigor especially in discrete mathematics included in the form of (a) new mathematics courses and (b) increased use of mathematics in other courses;
- a strong prerequisite structure which emphasizes the interdependence of the material;
- hands-on artifacts included to the extent possible in assignments and projects;
- inter-personal and engineering skills developed through laboratory and other projects;
- emphasis on use of knowledge and skills as they are acquired.

What Is The Impact Of These Changes?

The incorporation of these ideas into our teaching program has lead to a need for an entirely new undergraduate curriculum - courses designed to be more mathematically rigorous, more practice-oriented, more software engineering concepts and practice, more closely related to the real-world environment. This has necessitated a number of substantial changes to our program, specifically:

- extensive revision of many existing courses. All of the core courses containing elements of programming, data structures, machine representation, etc. will undergo extensive revision.
- development of several new course, especially in discrete mathematics and computation theory.
- the replacement of the programming language used for teaching purposes. C++ was selected as the new programming language and although not ideal, it has the benefits of supporting object-oriented programming and increasing industrial acceptance.
- a belief in the importance of reuse of courseware. We intend to make our artifacts available to other universities that choose to adopt elements of our curriculum.
- the development of the “closed laboratory” facility to support closed-laboratory exercises. The facility is being expanded to include devices such as motorized vehicles and other elements that simulate realistic applications.

What We Want To Share In This Paper

We offered the first course of our new curriculum (Introduction to Computer Science - 1CS) for the first time in Fall, 1992 and again in Spring, 1993. This first course has been enthusiastically accepted by our students.

What follows are some of our experiences and ideas about including software engineering concepts and activities in a closed laboratory setting as part of an introductory computer science course.

The course content and laboratory activities were developed under the guidance of Dr. James P. Cohoon. The laboratory activities were written by Dr. Cohoon, Messrs. Scott Briercheck, Dan Werbel, and Greg Fife.

How Can Software Engineering Happen In 1CS

What makes our first course (especially the laboratory component) of the curriculum revision different from other introductory Computer Science courses is our commitment to software engineering starting in the introductory courses. We have designed laboratory activities which deal with specific software engineering concepts and practices.

There are many aspects to software engineering. Techniques and procedures for developing, maintaining and improving software products are not easily learned nor easily adapted to one’s existing working approach. Many of these techniques and procedures can only be learned and understood by repeated expose and progressively more in-depth work.

Students should be made aware early on that “real” software products can be hundreds of thousands or even millions of lines long. They should have some exposure to what “real” software is - how it happened, what makes it good (or bad), why it’s important to have standards, how programmers and users interact with it, etc. We believe students can have meaningful software engineering experiences in introductory Computer Science courses.

There are many software engineering techniques and procedures students need to be exposed to and have experience with before they can develop a true sense of the discipline. To begin building this framework, we have included one laboratory activity which has students doing a code walkthrough, and another laboratory activity which has the students work with a large software system.

Both of these laboratory activities are discussed in more detail below. The exact text for each can be found in the appendix. Also included in the discussion below are some comments by students from evaluations of each laboratory activity. Although the comments are anecdotal, they give a sense of the attitude the students have for the laboratory experience and for the course.

Code walkthrough

Code walkthrough is a process described in every software engineering text. This technical review of some aspect of the software system can occur at any time during the development process. The focus of a code walkthrough is to review the material to date. “During the walkthrough the reviewee 'walks through' the material while the reviewers look for errors, request clarifications, and explore problem areas in the material under review.“ [3]

A simple text editor (miniroff) was written to demonstrate a code walkthrough. The idea of a code walkthrough was first introduced in lecture. Its concept and rationale are described in general terms. In the following laboratory sections, the TAs review the purpose of a code walkthrough, this time in relation to miniroff. The students also saw an overview of the function and organization of miniroff.

Groups of 2-4 students were formed and assigned a particular section of the program. Two weeks later, the groups were to make a presentation to the entire laboratory section on their portion of the code. A set of general guidelines on what information should be presented was included with the laboratory activity handout.

On the day of the presentations, the students are given a set of questions to answer - one question for each function (or section) of the code. These questions could all be answered if the student listens to the presentation and discussion. The TAs also have a list of questions (for reference) to help direct the discussion following each presentation.

The students believed this was a good experience. Their comments repeatedly stated working with their group was extremely helpful in understanding the code and that understanding and learning happened more quickly and easily. They did ask for more guidance with respect to the actual presentation and made several good suggestions which we will incorporate next semester.

When students were asked if they found working in groups useful in understand the code, they responded:

“Yes, it definitely helped because even if you get the basic idea of what the function is trying to do, someone else might pick something up that makes it that much clearer to you.”

“Yes, because the more you talk about the code the easier it is to understand it.”

“Yes! The items that I don’t understand immediately are given other viewpoints, such that the code is considered in more than one way, it will be easier to understand the programmers’s purpose.”

Large software systems

Once our students were exposed to the idea of a code walkthrough, we wanted them to have an opportunity to examine a large piece of software. Students do not have a realistic idea of just what large software system are. They can use them, but they have no concept how difficult it is to track through large amounts of code. They need to experience just how complicated and tricky working within large software systems can be.

We used a large software system called Fractint written by Messrs. Bert Tyler, Tim Wegner, Mark Peterson and Pieter Branderhorst. Fractint is freeware and the copyright is retained by the Stone Soup Group. Fractint calculates fractals and displays the results graphically. At 60,000 lines, Fractint is only about one-tenth as large as a typical inventory control system, but it is large enough to demonstrate some of the issues of software complexity.

Before the students came to lab, they were asked to respond to questions about their ideas on large software systems. The questions raised important issues about large software systems. For example, students were asked to respond to statements such as:

The most important attribute of a program is correctness. As long as a program runs properly, nothing else really matters.

As long as a program is properly commented, there is no need to create any other documentation concerning its[sic] design and internal organization.

In the lab, student groups of 2-3 first played with the software and executed its various options. They then picked one of three problems which forced them into the code.

They were not expected to be able to solve the problem. In fact, the objective was for the students to realize how complex large software systems are and how difficult it is to work within these systems even with other people helping.

Once the students seemed frustrated enough, the TAs lead them in a discussion on what factors made the problem(s) difficult and what might help make the problem(s) more solvable. The TAs were given a set of questions (for reference) which could help to get the students involved in the discussion and notes identifying important points of information.

When the students were asked to make a general comment on this laboratory activity, they responded:

“It was nice to do something that could help us realize situations in the ’real world’.”

“Good intro to large program management.”

“Made a point.”

"Very helpful in understanding importance of good programming organization before the actual code is written. Also, the importance of communication."

What's Next?

We are in the development phase of the second course of the new curriculum - Software Development Methods under the direction of Dr. John C. Knight. It will be offered for the first time in Fall, 1993. It will continue with an emphasis on software engineering concepts and ideas.

We also plan to offer a third/fourth year course which will focus in much more depth on software engineering. This course will begin development in the Summer, 1994 and will be offered in Spring, 1995.

Summary & Conclusions

We have recognized the need for software engineering to become an integral part of a professional computer scientist's skill and knowledge. One course in software engineering taken late in one's academic career will not provide the needed foundation and experience. With the development of the closed laboratory activities and the emphasis of software engineering throughout our new curriculum, we believe we are providing students with the necessary software engineering education to build, enhance and maintain quality software products for the multitude of modern computer applications.

The response and support of our students tells us we are on the right track.

Appendix

Bibliography

1. Communications of the ACM, v35, n11, November, 1992, p. 32.

2. Knight, John C., "International Perspectives in Software Engineering," The Rocky Mountain Institute of Software Engineering, Boulder, CO, Q1, 1993.

3. Fairley, Richard E., Software Engineering Concepts, McGraw-Hill Book Company, 1985, p. 186.

Curriculum Committee Members

Dr. Wm A. Wulf
Dr. John Knight
Dr. Randy Pausch
Mr. Greg Fife
Ms. Sally McKee*
Dr. James P. Cohoon
Dr. Worthy N. Martin
Dr. Jane C. Prey
Mr. Rob Deline*

*Student representatives