A PRACTICAL UNDERGRADUATE INTRODUCTION TO SOFTWARE ENGINEERING

John Foreman

Texas Instruments, Inc.
Lewisville, Texas

ABSTRACT

Much has been written about proper software engineering methods and how to teach these techniques to students. Several authorities have analyzed our present techniques as needing significant improvement [1]. This paper discusses a practically oriented introduction to software engineering aimed at introducing students to sound development habits and life-cycle thinking early in their careers.

INTRODUCTION

Computer Science 380, Software Engineering Fundamentals, is a 1 semester, 3 hour course taught at the U. S. Air Force Academy (USAFA) whose rather general purpose was the initial transition of computer science majors into computer professionals. During the Fall 1979 and Spring 1980 semesters, during which I directed the course, topics covered included: program development in an interactive environment, advanced programming, COBOL, static and dynamic data structures, algorithm development, and software engineering concepts and principles.

The students in this course were predominantly sophomores or juniors, had at least taken the compulsory introductory Computer Science course, and might concurrently be enrolled in an architecture course. Enrolled students had programmed in Burroughs Algol in the compulsory introductory Computer Science course; Algol and Pascal would be used in subsequent courses. COBOL usage in the military was the prime reason for its use here. In my opinion, COBOL did not cause any major problems to the students; one of the course goals was for the students to understand and deal with the control structure deficiencies, and utilize good development practices. (Note: This paper will NOT address grading policies or techniques regarding the programming exercises. Due to the administration of the Cadet honor code at USAFA during the time of this course, it was only possible to grade optional exercises, the design review and demonstration associated with the final project.)

The most important part of the course was the programming exercises which were especially designed to illustrate the implementation of selected data structures, the importance of the design phase of the software life cycle and software engineering considerations. Especially important was the idea that programs have a life and are not to be programmed and thrown away [1]. The rest of this paper will discuss the exercises used in the course.

EXERCISE 1

Exercise 1 was a control break type problem, which produced a military "Status of Forces" listing [2], and was primarily used to get the students somewhat familiar with COBOL statements and control structures, data division capabilities and proper design. The interesting twist for this exercise was that after each student successfully ran his program against the problem data set, they were required to re-run their program against another data set, identical in format, but different in contents. The intent here was to illustrate to the students the general applicability of algorithm and to identify the dangers of "hard wiring" in the code.

EXERCISE 2

Exercise 2 was a simple payroll type application, where the required outputs were to compute regular pay, overtime pay, federal taxes, social security taxes, take home pay, etc. This exercise was the vehicle to introduce COBOL array processing since an employee master file containing pay rates was read in and had to be searched for each weekly time card to be processed. Especially important here was the idea of algorithmic isolation, especially for the sequential search, since the transactions to be processed included employees who did not exist and for whom error messages had to be output. A classic problem here for the students was the idea that the search function returned a found or not-found indication, which then dictated the remaining processing to occur. Many of the students attempted to compound the search function with what was to be done when the individual was found, and had lots of problems handling the not-found situation.

Exercise 2 also introduced another facet of the course, optional bonus exercises. These "optionals" would be included on all future exercises and were designed with the intent that if the student had done a good job of modularizing his design, the placement of the new functions was somewhat trivial. Indeed several students stated: "the optionals just sort of fell right out; is that what you intended?". The optionals also allowed a student to potentially raise his grade in the course, and many availed themselves of this opportunity. The students were encouraged to design their solutions with the optionals in mind, but delay implementing the optionals until after the main exercise was completed. In this way we hoped to encourage some long-range thinking.

EXERCISE 2A

Exercise 2A occurred about 2 weeks after the completion of Exercise 2, and was NOT written up in the course syllabus. The intent was that the students NOT know it was coming; and indeed this was true except for those who talked with students from the previous semester. Before I discuss this exercise let me say that by this time in the course the students had been exposed to programming standards (adherence to which was required), were familiar with top down design, top down implementation, stubbing and top down testing. Exercise 2A required the student to complete a series of modifications to exercise 2; these modifications specifically addressed array and record processing. As you can probably imagine, the thought of having to modify a program did not thrill anybody (some knew their code was not clean -- I especially knew it from debugging much of it), but it is an important lesson for a student to appreciate. The exercise was made all the more practical by the requirement that the modifications be accomplished on another students program. Indeed each student was told who to change with (A did not exchange with B); instead A gave his program to B; B gave his program to C, etc.) and was forbidden to discuss his code with the modifier. This was designed to simulate the real life conditions of personnel turnover, which impact so heavily today. As a last requirement, each modifier had to write a critique of the program he modified.

The consequences of this exercise (other than the mumbling of many 4 letter words) should be obvious. Indeed perhaps the most interesting surface effect (other than an appreciation for the real world of software development) was the improved quality of the listings on subsequent exercises. The indentation improved, the variable names were much more significant, the modules and paragraphs were more single function oriented, and it was evident that more up-front planning had been done. (I often wonder if the students were just trying to protect themselves from another "exchange and modify".)

EXERCISE 3

Exercise 3 was a message encryption/decryption exercise [2] which required the students to utilize strings and stacks. An extensive set of optional exercises was included.

EXERCISE 4

Exercise 4 addressed linked lists by requiring that an airline reservation system be implemented. The functions were add a passenger to a flight, delete a passenger from a flight, list all persons on a specific flight, add a flight, delete a flight, etc. Examples of such a system can be found in [3]. This exercise provided a natural means of introducing the COBOL case construct (go to depending on) and re-emphasized the importance of early design and modularity. Transactions were included for every possibility, including non existent flights, individual flights being full, no available space left in the link list, incorrect transaction types, non existing passengers, so the students had to be prepared. It is interesting to note that the sequential search required in Exercise 2 had to be used again in both Exercise 3 and 4, further reinforcing the idea of single function modules. A poorly written search from Exercise 2 had to be "stripped down" to re-use it in Exercise 3 and 4.

Exercise 4 also required that the students work in teams of 2 or 3, as designated by the instructor, so that the students were exposed to the further problems of having to develop software in a group. The idea here was to show that if the design and interface specification was done correctly work could proceed independently. Additionally, it familiarized the students with the problems of communication and coordination inherent in software development.

TERM PROJECT

The term project was again a team exercise and required the students to use the exercises they had created during the semester. The requirement was to link the "status of forces", encryption/decryption, and airline reservation system into an interactive user oriented system. This required that a driver/menu program be written which allowed the user to select which capability was desired, and the binding utility be utilized. Additionally, modifications were required in each of the component programs. This approach worked well with the team concept. The objectives of this exercise were many: re-use of existing software, team-work, program modification, use of the binder, interactive programs, and user interface.

Requirements for completing the term project included a 1 hour design review and an interactive demonstration by each team of their system. Students were given free rein as far as the capabilities their system provided; several produced help facilities, while others utilized an instructor supplied file of Burroughs TD822 terminal control characters in order to provide reverse video, flashing, and other terminal control capabilities.

The design review addressed the major areas of each group's overall design and implementation plan, work breakdown and allocation, and individual team member responsibilities. Requiring such info required the teams to have agreed on these areas; additionally, it greatly curtailed the student tendency to "wait 'til the last minute". An added benefit was that each student had an opportunity to practice some communications skills. The design reviews were attended by other faculty members to make them even more realistic.

The demonstration provided an excellent opportunity to quiz students about the internal workings of their software, and some basic architectural questions. The importance of user interface was emphasized by occasionally having one of the department secretaries run the products, and by also throwing "garbage" input against the system.

CONCLUSION

While unquestionably a significant effort was required from the students in this course, they were challenged, for the most part learned a lot and generally had fun doing it. This was especially true on the term project. The course met its content objectives and in several cases caused some marginal students to re-evaluate their choice of major.

Academic programming education should be the training ground for the development of proper design and engineering practices, and provide insight into real world production software, in addition to teaching algorithms and data structures. There is plenty of opportunity to achieve this goal in just about all courses; hopefully this course can provide a model for future efforts.