# Requirements for Student Programs in the Undergraduate Computer Science Curriculum: How Much is Enough?

Lionel E. Deimel, Jr.
Mark Pozefsky
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27650

ABSTRACT

Student-written programs accepted by computer science instructors are usually inferior to programs which exemplify currently-accepted "good" professional practice. Although enforcing more rigorous standards for programs places an additional burden on students and faculty alike, substantial benefits may be gained thereby. The nature and implementation of such standards are discussed.

## A Dilemma

A recent survey of B.S. graduates from the Computer Science Department at North Carolina State University elicited numerous comments about the quality of the undergraduate curriculum at North Carolina State University. One particularly striking remark was that programs routinely accepted by Computer Science instructors would not be considered even minimally acceptable in an up-to-date programming shop. The person who made this observation clearly felt his training in the Computer Science Department was unrealistic. On the other hand, one hears pleas for mercy from current students, who claim that demands for carefully designed, well-documented programs disrupt their lives with sleepless nights at the Computing Center. These students do not exaggerate, yet neither does the graduate--the instructor is faced with a difficult decision: how much can reasonably and profitably be expected of student-written programs?

The dilemma is actually somewhat more complex. With the growing recognition of the importance of program structure and documentation, there has developed a consensus that programs have to do more than just "work." In the first and second programming courses, students are required to produce better designed and documented programs than they were a few years ago. Quite specific new standards are being imposed. These standards are less likely to be imposed on programs written for data structures, numerical analysis, or simulation courses, where improved programming skill is not a major objective. What stance should instructors adopt with regard to programming assignments in these courses where programming skill is, in a sense, incidental?

## What Can We Ask?

Let us first characterize what we might like to require of student programs. The primary requirement for a program is to accomplish the function outlined in the assignment specification. The program should be fully debugged and tested. Hence, it "works" for all input in the manner specified by the problem assignment. Ideally, programs are required to produce "correct" output for "good" data and output meaningful diagnostics and possibly implement recovery procedures for "bad" data.

The actual source code should be modular and use structured control constructs in a way that reflects the logic of the task being performed. The programming language should be used intelligently. (PL/I programs should not use precisions in variable declarations that are not "natural" for the host machine, for example.) The code should be documented to clarify what is being done and should be formatted in such a way as to make the structure clear. Each module (procedure, subroutine, etc.) should be preceded by a prologue of comments providing such information as module name, programmer name, date written, purpose, parameters (if any), algorithm description, error considerations, and input/output formats. All variables should be declared, and their exact functions spelled out. Non-obvious constants should also be explained.

For certain assignments, particularly in advanced courses, additional requirements beyond the production of a program may be imposed. The student might be required to produce user documentation, additional technical documentation, or associated job control procedures.

The general case for such program requirements is adequately detailed elsewhere [3]. These requirements have come to be accepted as part of good programming practice. But is it practical and effective to demand such high standards of undergraduates?

## The Case for Moderation

Writing programs is very time-consuming. Few people who have practiced the art of programming will dispute this statement. Furthermore, program writing as a course assignment is different

in a significant way from doing physics problems. The computer monitor's a student's work in a way that will not allow him to submit erroneous work while maintaining the appearance (or the illusion) of its being correct. The computer points out mistakes, thereby pressuring the self-respecting students to do more and better work than he might otherwise. This fact is in large measure responsible for the student complaint that programming courses carry too few credit hours. Programming must be practiced to be mastered, however, and the complaint appears inevitable. If a student feels a programming course is unduly burdensome in relation to its credit load, perhaps he is pursuing the wrong major. Nonetheless, a single instructor cannot claim all a student's time--asking for all conceivable documentation and program generality may be excessive. We must not convert a heavy burden into an impossible one.

Brooks, in his collection of essays The Mythical Man-Month [1], makes the distinction between a program and a programming product. The former entity is the minimal requirement of a programming assignment--a program which can be run by its author to perform some function. It differs from what we would like students to produce in most courses -- a programming product, a program which can be run, maintained, and extended (perhaps by other programmers) and is generalized enough to be useful under a variety of conditions. Alas, Brooks estimates a programming product requires three times the effort of a mere program. In other words, requiring more niceties is equivalent to tripling the size of a programming assignment. This observation should give the instructor pause. After all, few student-written programs ever become production programs, and many are written only to illustrate a particular technique or language feature.

Undue emphasis on niceties particularly in early courses, may even be counterproductive. Certainly the student should not be led to believe that it is unimportant that his program does not "work" so long as it is constructed in accordance with a particular set of rules. Yet this impression is often conveyed. It leads to "pretty" programs which are meaningless and to over-documentation of programs, which obscures rather than explains.

In answering the question of how much is enough, we should realize that a program to solve nearly any nontrivial problem can be expanded and perfected almost without limit. Instructors should not require unreasonable generality in programs (for example, requiring extensive input validation for a simple program being written to illustrate the use of a particular language construct) and should help their students recognize the point of diminishing returns when perfecting a program.

Why We Should Still Ask for a Lot

The potential benefits of being more demanding are substantial. They include giving the student a better understanding of what computer professionals do, increasing his effectiveness as a programmer, and making his investment in program writing pay long-term dividends. We feel these benefits outweigh the arguments for accepting less in student programs than would be required by industry. This conclusion means, of course, that both students (as programmers) and instructors (as graders) need to work particularly hard in computer science courses.

Perhaps the most compelling reason for requiring extensive program documentation and generality is to give the student a taste of the real-world. Ultimately, it is not fair to send the student forth from academia naively believing he will forever be asked to write modest little programs for his personal use. Instead, he will spend much of his time maintaining and extending old programs, writing software to be used by non-programmers, and producing technical documents. The student deserves to know these facts of life so he can make an intelligent career decision. He will be better served if we prepare him to do well in the environment he is likely to enter by making realistic demands upon him while he is in school. The purist-scholar, of course, may argue that only great principles should be taught at a university, that certain details of implementation should be left to industry and to trade schools. We have, after all, flattered ourselves with the title "computer science." To this we say that in fact most computer science graduates will first become programmers, and we bear an obligation to help them become competent programmers. Realistically B.S. computer science graduates are no more "computer scientists" than B.S. physics graduates are "physicists."

Requiring something beyond Brooks's program and something more like his programming product is in large measure a matter of making one's programs more user-oriented. It is a real problem that because of the nature of the academic environment, students tend to write programs for themselves intended for a single use. Most real programs are not of this nature, but rather are written by programmers for other users. Requiring students to produce meaningful error messages, to validate input, and the like makes them realize that they will not generally be the ultimate users of their software and encourages them to consider the needs of those ultimate users.

Another reason for requiring a lot is that many of the demands we are speaking of represent important techniques for increasing programmer effectiveness. The student who has to comment what a variable represents must precisely determine for himself what it represents. If the instructor refuses to accept the statement that "I is a pointer," the student will have to clarify in his own mind what is being pointed at. Fuzziness in this example leads to the "off-by-one" syndrome: does I indicate the last used array location or the next available one? Errors in program design and coding decrease when such details are explicitly confronted. If standards are enforced, the student is more likely to recognize such alternatives and avoid confusion between them.

Although student-written programs do not generally graduate to production status, this fact

does not inevitably lead to the conclusion that they need not conform to good programming practice. The role of the one-time program--the code which is written, run once, then thrown away--is very limited. One-time programs have a way of being picked up later and generalized or incorporated into other software. This statement does not usually apply to student programs, however. They often truly are one-time programs which are discarded after they are graded. This is a shame. To be sure, many student programs thereby get what they deserve, but the student is not well-served. The rarity of one-time programs should be pointed out to him, and he should be encouraged to retain his programs. They may be of future use to him in several ways.

Students usually retain their notes taken in college courses. Many students save their textbooks as well, and those who do not often wish they had. The reason for this is that such materials serve as excellent reference sources in future years because of the association the student has had with them. Not even the most brilliant programmer can remember everything, so he frequently consults old programs to see how he has handled certain problems in the past. He can do this only if (1) the programs have been retained and (2) the programs remain readable long after they have been written. We should encourage students to treat their programs exactly as they do class notes. Programs should be bound and saved for reference. This is another reason for requiring good structure and documentation in student programs, as retention will do the student no good if the programs are not readable.

User-oriented programs are usually generalized programs. By requiring such programs, we get the student to think of a program more as a function which transforms valid inputs into valid outputs. This, in turn, will help a student recognize that generalization in his subprograms can allow them to be utilized in future programs. That is, he is, or can be, encouraged to build and use his own program library. This again is possible only if programs are retained.

We now must consider how our observations affect the way we teach computer science courses.

## Implementation within the Curriculum

Program requirements should no doubt be most strict in early courses, generally the first two programming courses. If good habits are to be inculcated in the student programmer, it is best to begin the indoctrination from the very start. Continuity of standards needs to be maintained throughout the curriculum, however. If this is to be done, the faculty needs to reach a consensus that standards are necessary and must decide what it considers important. This will no doubt represent a thorny problem for many departments, as programming style remains highly individualized. Perhaps a certain measure of faculty independence may have to be surrendered in order to develop shop standards for a department.

In the first course, standards must be implemented incrementally, since it is impossible to teach enough fine points of the art of programming to allow each assignment from the first to be covered by the same set of standards. Structured programming and internal documentation should be taught from the beginning; more subtle issues such as the use of Boolean rather than integer flags can be introduced later. It is important, however, not to cut too many corners just to get students running programs quickly. Most importantly, as standards are introduced, they must be strictly enforced. In fairness, this means they should be written down, so far as possible. Enforcement, alas, requires reading programs line-by-line. The instructional support for this is sometimes lacking, although it is a task which might be successfully carried out by carefully screened upperclassmen or graduate students. We feel that certain needs of students can be fulfilled in no other way. In particular, students learning to program in an incremental way (learning half-truths in order to allow them to begin writing programs) tend to fabricate misconceptions about the environment in which they are working, especially when the computer is initially treated as a black box. Student programs reflect some of these misconceptions. For example, a student may initialize all variables, even those whose values are immediately input to the program, so that the computer will not have "undefined values" within it. Such an error (and misconception) would go unnoticed if programs were never carefully read but only checked for correct output. The misconception becomes an article of faith, persisting long after the student's training should have destroyed the myth. Other techniques can also be used to encourage good habits, such as distributing copies of good programs (from the instructor or one of the better students), exchanging student programs for comparison, and so forth. In general, reading of programs should be encouraged, just as general reading is encouraged by English teachers.

Conveying program design philosophy is particularly difficult at the beginning, when the student has little experience with computer software. Emphasizing that the compiler is just a program can be of great help here. The compiler was written by programmers for other users, and those users (students in this case) are not interested in how it works, but <u>are</u> concerned about receiving meaningful diagnostics. Generality and intelligent error handling can be encouraged by citing this model. Testing programs with data the student have not seen helps enforce requirements for user-oriented features [2], and having students exchange programs for testing improves student understanding of the need for such features.

The usual two programming courses taken by computer science majors are insufficient to develop real fluency in programming. If only for this reason, we believe that standards should not be relaxed as students enter middle-level courses utilizing programming assignments. Instructors in such courses (for example, data structures) should still examine programs thoroughly and severely penalize documentation lapses and clumsy constructions. (Certainly the English Department would not drop its requirements for proper spelling in themes for middle-level literature

courses.) The arguments for asking a lot are equally important here, particularly those relating to increased programmer effectiveness and notes for future reference.

In upper-level courses, we again believe standards should not be relaxed. Their enforcement, however, can be. If a student has had the discipline we recommend, following the standards will be second nature by the time he is a senior. Certainly large programming projects need not be read line-by-line. Spot checks can reveal generally poor programming technique and reading module prologues should make the overall program structure clear. Projects of this nature usually require user manuals to be written, and these should be read carefully. By the time the student reaches this level, he should have gained the maturity to recognize the value of the requirements imposed upon him. Failure to use the techniques we advocate will result in less effective programming, particularly where programming teams are involved. The symptoms of this failure are late programs and programs which do not perform according to specifications. Students will tend to use the methods we have tried to teach them if only in self-defense. Thus, the standards in large measure enforce themselves.

## Summary

That computer programming courses consume more time than other courses carrying equivalent credit is unfortunate, but we feel there is little that can be done about it. On the contrary, the disparity which now exists between what is required of programs in school and what is required in industry should not be tolerated. Although this point of view requires more work on the part of both students and faculty, the end result of its implementation should be more effective programmer training.

## References


1. Brooks, F. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, Mass., 1975.

2. Deimel, L., and Clarkson, B. The TODISK-WATLOAD System: a convenient tool for evaluating student programs. Proc. 16th Southeast Regional ACM Conf., 1978, pp. 168-171.

3. Yourdon, E. Techniques of Program Structure and Design. Prentice-Hall, Englewood Cliffs, N.J., 1975.