Developing an Introductory Software Programming Course for Engineering Students

Scott J. Schneider
Department of Engineering Technology
University of Dayton
Dayton, OH 45469
sschneider@udayton.edu

Abstract

The ability to effectively develop software programs, from complex software systems to simple macros, is becoming increasingly important in all engineering disciplines. Educators have realized this need, and likewise have included software programming in many engineering curriculums. The initial course in software programming has historically focused on learning the syntax for a single programming language instead of the skills of logical and algorithmic thinking and the processes for software development. This paper presents a stepped process for introducing software programming to engineering technology students.

1 Introduction

Working as a contract engineer for numerous companies has allowed me to interact with both young and veteran engineers developing software systems for a myriad of industries. This experience made evident the shortcomings of my software programming education as well as that of many of my peers. While I was competent with the syntax and structure of programming, I was ill prepared to tackle large problems or complex systems. My deficiency was in understanding the software programming process. Those colleagues that obtained an education in computer science were much better prepared to tackle software design using proven techniques than their engineer counterparts. The main difference is the "code it first" mentality that many engineers have when it comes to software development.

The "code it first" philosophy arises from both a lack of knowledge about the software development process and only being introduced to software programming courses that focus on developing the syntax skills of programming. During my time in industry, it became evident that one's ability to implement a structured software development process is just as critical as one's syntax skills. In moving to an academic environment, I once again confronted the "code it first" mentality.

Computer science and engineering educators have long realized the importance of providing engineering students with a solid understanding of the software design process [1]. However, the first course, and often times only course, an engineering student receives in software programming is typically based on learning a particular programming syntax with little emphasis placed on understanding the software design process. In teaching an introduction to software

programming course, I intend to emphasize the software design process, not just the syntax of a particular language.

During my first semester teaching the introductory programming course I recognized that not only do students lack understanding about software design, but they also face a deeper problem: not understanding the logical and algorithmic thought process. Software engineering is a discipline about solving problems through the use of computer assistance. Students should master the concepts of logical and algorithmic thinking, including the ability to break analytical problems into their logical elements and then to create a sequence of actions for the realization of a solution.

While the current introductory course is providing students the necessary tools to develop software programs of problem solutions, it does little to aid the student's ability to actually develop the solution. Through repetitive coding exercises some students develop an ability to solve problems algorithmically, however many never achieve a full ability to do so. Instead of leaving this critical skill development to chance, the development of a student's ability to think in a logical and algorithmic manner needs to be directly emphasized.

2 Background

In the Engineering Technology Department at the University of Dayton an introductory software programming course ECT361 is offered, which is taken by students in several engineering technology disciplines. This course is the only software programming course offered from the engineering technology department. This course has historically been offered as a C++ syntax course and has a history of being very challenging to all enrolled students, especially those students not coming from the electrical or computer engineering disciplines. The students from the electrical and computer engineering technology majors have already had a digital logic course which has provided them some insight into the logical and algorithmic thought processes before entering the class.

For many of the students, the key outcome from the course is not mastery of the C++ syntax, but rather the ability to develop algorithmic solutions to problems that can be implemented in software using a software design process. Therefore, the two goals in redeveloping this course are to explicitly focus on developing the students' logical and algorithmic thinking capabilities and their understanding and application of the software design process. The students leaving the course will be able to break a problem into individual components and sequence them together into an algorithmic solution that they can follow through a systematic design process to the final coded implementation.

2.1 Historical Perspective

Trying to define the best method for introducing students to software programming is not a new problem or one that is likely to have a single answer. As the discipline of computer science evolved, educators focused on ensuring that the science of software programming was paid the same attention as in other science disciplines. The science of software programming starts with the logical and algorithmic thought process. The goal is to teach students how they can solve any problem by finding an algorithmic solution to it [2]. Even though the science of programming is well understood, the discipline evolved from being considered merely a tool [3]. Therefore, the

focus on learning programming syntax instead of algorithmic thinking or software design has a long history within the computer science discipline. While programming languages and styles have changed, the same basic problems in teaching the science of software programming still exist.

The shortcomings of a programming-first approach in the introductory curriculum is outlined in the ACM Computing Curricula 2001. By focusing on software programming solely, the students are provided a narrow exposure to the computer science discipline which tends to oversimplify the design, analysis, and testing activities. The understanding of syntax with inadequate algorithmic skills leaves students with an improvised method of program development, often relying on trial and error. However, the programming centric introductory courses do allow students to acquire new programming skills in a much applied manner [3].

Most of the research material on introductory programming courses has focused on courses for students majoring in Computer Science. Therefore, the incorporation of problem solving and algorithm development and the software design process into the curriculum often takes place over a several course sequence. For engineering students who have room in their course load for a single software programming course, these beneficial elements are typically left out or glossed-over in order to maximize the time spent on learning the syntax of a particular software language. A syntax only course risks providing a hollow service to the students, especially for those who do not utilize the language offered in their particular programming course when in industry. Even those students that do use their programming skills in industry are left to learn about the software development process in an industrial, non-structured, non-scientific environment which can often lead to poor habits.

2.2 Software Programming Educators Survey

A survey conducted of engineering technology professors in the United States that teach an introductory software programming course revealed numerous opinions and methods for providing this training. From the responses received, a wide variety of methods and opinions in how to best teach an introductory software programming course were noted. Responses ranged from those faculty members who have developed a multiple course sequence with an introductory course solely focused on teaching algorithmic development, to those who use either an academic programming language or one which is easier to grasp, to those that have come to a realization that students are unable to be taught the fundamental logical or algorithmic thought processes. The greatest success seemed to come from the respondents who have a pure logics and algorithms course in their curriculum.

In our Engineering Technology Department we must provide a sufficient proponent of syntax in the course since this course is the only one that many of our students receive in software programming. Having a hands-on and applied curriculum precludes us from a strictly logic and algorithm development course. The ability to break problems apart and develop algorithms is an important outcome. However, our students also need to have a mastery of the code writing and debugging processes within the software development design process.

Another topic that came up within the survey was on the poor choices for text books available for introductory programming courses. Many respondents referred to a lack of good syntax

based books that provide a sufficient amount of material related to program design and algorithmic development, leaving dedicated professors on their own to develop the material. Some good material does exist, however not in all programming languages, or sometimes in a syntax-independent fashion which serves primarily the Computer Science Departments.

3 Course Methodology

In an attempt to provide a fuller learning experience to our introductory software programming course ECT361, several changes were made to the curriculum. These changes are being introduced over a two semester period. The fall 2004 semester had an extensive software development component added. The emphasized logical and algorithmic thought process development is being introduced in the winter 2005 semester. The new course outline is:

1. Logical and algorithmic thinking and problem solving
2. The programming environment
3. Variables and storage classes
4. Keyboard input and screen output, input and output formatting
5. Lexical elements and operators
6. Arrays, pointers, and text strings
7. The software development process
8. Control structures and logical operators
9. Functions, recursion, references, scope
10. Formatted input and output from files

Even though the C++ syntax proves to be more difficult to master than other popular programming languages, the course retained the C++ language for this course given feedback from the Engineering Technology Department's Industrial Advisory Committee. Being the sole programming course, the students do need an appreciable experience with a common programming language that they may encounter in industry. An important aspect of the students' learning process is the ability to implement the algorithms that they develop [3].

3.1 Incorporating Software Development

A goal in revamping the ECT361 curriculum is to retain the same level of mastery over the C++ syntax as was covered in the previous course. Providing adequate time for a sufficient introduction to the program design process without sacrificing the syntax content covered is challenging. The focused software development learning is introduced before and in conjunction with control structures since the non-sequential nature of control structures is known to be a difficult topic for students to master. The students' ability to separate and tackle the problem independent of the syntax would aid in their mastery of the material, allowing them more time to focus on learning the syntax.

While programming design is taught through exercises primarily in conjunction with learning the syntax for control structures, the design philosophies are all introduced prior. Requiring that

students perform program design does not lead to true design practices by the students. However, outlining the design process and highlighting it conceptually using real-world problems can provide enough of a solid foundation to cause students to understand the benefits from committing to the full design process [4].

The formal processes that exist for software development may take several forms and typically include the key steps of problem definition, solution outline, algorithm development and analysis, and conclude with code writing and testing. Following such a process allows the engineer to focus on developing the algorithms without worrying about code syntax, and likewise allows the software syntax to be created without concern for the correctness of the underlying algorithms.

Standards exist for program design, including the use of the Unified Modeling Language (UML). The standard notations for activity diagrams and the general principles for developing algorithms using pseudocode are provided to the students as references. Since program development is a personal experience, and often non-standards based methods are used in industry, I feel that the process is more important than the method. Students are, however, required to perform three key steps during their program development process: break the problem into its respective inputs, outputs, and processes, outline the algorithm, and finally test the algorithm.

The best method found for demonstrating the importance of program design to students is through the use of class-participation programming exercises. During this process, students work with the instructor in solving a problem through the creation of a software program. The problems are tackled both with and without going through the software development process to highlight how the time spent in preparing to write the code does save time and produce cleaner, more efficient code. One key aspect demonstrated to students is that catching and fixing algorithmic errors is much easier if done prior to the code writing phase. Once the algorithm is coded it is often difficult to debug the syntax and algorithm independently to determine the root cause of a problem.

A full program development process is required for all homework and project assignments that result in a code writing exercise. Providing a consistent emphasis on the program development process is essential. The consistent emphasis on program design throughout the later part of this course ensures that every student has had ample time to develop their own strategy for breaking problems apart and creating algorithmic solutions to them.

3.2 Teaching the Algorithmic Thought Process

Some educators are unsure of the ability to teach logical and algorithmic thought processes. I feel that these skills can be learned and an attempt should be made to help foster them early in the course for maximum benefit. In teaching the program development process, the student's ability to logically break apart problems and develop algorithmic solutions did increase. However, given the diverse technical background and previous programming experience that our students have, it is still necessary to implement a focused algorithmic development component to this course. Research demonstrates a direct correlation between a student's experiences with software programming to their performance in an introductory programming course [5]. If students are able to focus on developing their logical and algorithmic thought processes early-on,

it stands to reason that they would be able to focus more directly on the syntax during the remainder of the course.

Learning a computer programming language is not the same as learning a foreign language, though often they are taught in a similar manner. With foreign languages, the syntax is often emphasized since, as humans, we already know how humans think and therefore, understand how to converse with them regardless of the language which they speak. However, before trying to talk to a computer one must first understand how a computer thinks to enable one to communicate with it effectively. The students' ability to develop algorithmic processes is critical at this point.

The first couple weeks in this course are spent honing the students' problem solving skills. The primary purpose is to force students to think in a logical manner similar to how a computer needs to be commanded to complete a given task. In order to achieve maximum utilization from this development time, the programming environment that is used for the course is also introduced. Students are required to provide their algorithmic solutions in the form of commented code. This added step, while slightly cumbersome for the students, allows them to use this time to become familiar with the coding environment that they will be using later in class. This process also has the added benefit of helping the students to feel like they are still accomplishing something and not just performing busy-work.

4 Outcomes

The inclusion of the programming design process into the course curriculum resulted in no loss in syntax material able to be covered. In fact, several concepts were able to be investigated more fully since the students are better able to directly focus on the syntax. Even though the difficulty of the syntax learned after the introduction to the program design process increased, the class performance rose as indicated by a rise in their cumulative grades. Student surveys indicate an overwhelming support for the benefits of using a software design process. Most notably, in the final project, students are asked to develop a complete software program including complicated programmatic and mathematical algorithms. Several students explicitly commented on the benefits of performing the program design prior to starting the code writing. One student commented "It proved that the development process is necessary, especially with programs that make use of more complex algorithmic processes."

Even though time is spent to define the design process, and students seemed to "buy-in" to the benefits of it, some students developed their design material from the code instead of writing the code from the design. Reviewing how other instructors have solved this problem, I decided to administer two homework assignments, one for the program design and another one for the code. The code homework will be collected one week after the program design [4].

This course is offered twice a year and data is being collected to monitor the impact of this methodology on students' success in this course and in their logical and algorithmic capabilities as well. Preliminary results to a prepared survey focusing on students' self perceived difficulties with the course after the fall 2004 offering indicate that the students overall feel more confident with their problem solving capabilities than those students from the previous offering where the course was taught using a syntax only method. However, performance disparities still exist

between students from the electrical and computer engineering technology programs and those from the other engineering technology disciplines. The direct focus on logical and algorithmic thinking in the beginning of the course will help reduce this disparity, allowing those students outside the electrical and computer engineering technology programs to catch-up.

5 Conclusion

The question of how to teach an introductory software programming course is not new, and does not have a single, easy answer. Though most of the research addressing this issue is focused on the computer science curriculum, it provides valuable methods for achieving optimal coverage in an introductory course for engineering students. The major goals in developing a new curriculum stem from an awareness of the lack of preparation of engineers entering industry to affectively perform software development. Initial feedback from students has shown awareness to the benefits of following a structured program development process. Another problem in the course results from previous offerings showed a poor performance among the students in programs outside of electrical and computer engineering technology. A new course curriculum focused on teaching logical and algorithmic thought processes has yet to be evaluated, but is intended to help mitigate this problem.

Bibliography

[1] W. Hankley. Software Engineering Emphasis for Engineering Computing Courses. *Proceedings of the American Society for Engineering Education Annual Conference*, document 2305, June 2004, Salt Lake City, UT.

[2] D. Gries. What Should We Teach In an Introductory Programming Course? *Proceedings of the fourth SIGCSE Technical Symposium on Computer Science Education*, 6(1), pages 81-89, 1974.

[3] Computing Curricula 2001, Chapter7 Introductory Courses, http://www.computer.org/education/cc2001/final/chapter07.htm.

[4] A. Ghafarian. Teaching Design Effectively In The Introductory Programming Course. *The Journal of Computing in Small Colleges*, 16(2), pages 203-210, 2001.

[5] E. Holden and E. Weeden. The Experience Factor in Early Programming Education. *Proceedings of the 5th Conference on Information Technology Education*, pages 211-218, October 2004, Salt Lake City, UT.

Biography

SCOTT J. SCHNEIDER is an assistant professor of Electrical and Computer Engineering Technology at the University of Dayton. He received his MS in Electrical Engineering from The Ohio State University. His areas of interest include software development, embedded systems, and automotive technologies. He has designed and implemented advanced embedded systems for the communications and automotive industries.