Impact of Preprogramming Course Curriculum on Learning in the First Programming Course

Rob Faux, *Member, IEEE*

***Abstract*—This paper reports the results of research that measured the value of integrating problem solving, algorithm development, pseudocode, and diagramming techniques into introductory computer science courses. The hypothesis was that the introduction of these topics prior to the introduction of a programming language would reduce the learning curve requirements and increase the success rate for beginning programmers. Posttest programming scores for like tests exhibited some difference within the sample, with the treatment group performing better than the control group. Members of the treatment group reported a better experience in the introductory course and saw more applicability for that course than members of the control group. In addition, treatment group members utilized pseudocode more consistently and applied the idea of drawing pictures to aid in writing code during their posttest experience. The combined results indicated that learners, in general, found that the introductory course contained useful content and that they could successfully apply processes learned in that course as part of a programming strategy.**

***Index Terms*—Breadth-also approach, breadth-first approach, CS0, CS1, depth-first approach, diagramming, introductory computer science, problem solving, programming learning, pseudocode.**

I. Introduction

THIS project measured the value of integrating problem solving, algorithm development, pseudocode, and diagramming techniques into introductory computer science (CS0) courses. For the purposes of this research project, CS0 was defined as an introductory computer science course with no programming language focus. The CS1 course was defined as the first programming course. This structure is typical of most breadth-first curricular approaches. An existing computer science program at a mid-sized university in the midwestern United States, which was utilizing the CS0/CS1 breadth-first model, agreed to implement curricular changes intended to refine, increase, or add coverage pertaining to problem-solving concepts, algorithm development, and algorithm testing. Data collection tools were implemented to measure the impact of these curricular modifications in CS0 on programming skill learning in the subsequent CS1 programming course. All data was collected during the CS1 course for control (no new curricular material) and treatment (new curricular material) groups. The design of the study attempted to triangulate the data by supplementing the traditional "treatment/posttest/posttest" design with self-evaluation ratings, course satisfaction ratings, qualitative testing data, and qualitative entry/exit survey questions.

Manuscript received February 25, 2004; revised May 7, 2005.

The author is with Computer Science and Mathematics, Wartburg College, Waverly, IA 50677 USA (e-mail: rob.faux@wartburg.edu).

Digital Object Identifier 10.1109/TE.2005.852593

II. Introductory Computing Curriculum

Current guidelines for the introductory sequence in computer science programs consist of multiple models [1]. The variety within the guidelines reflects the uncertainty felt by many computer science educators regarding which approaches serve learners best. Unfortunately, the current body of knowledge in computer science education is relatively new and largely incomplete [2]. Other than a growing source list outlining curricular options, supporting arguments, and anecdotal results, research data is insufficient to support any one approach as a best practice.

One of the subproblems that has drawn much speculation and interest is how computer science disciplines can best support programming learning within the curriculum. Current approaches often fail to reach a broad base of learners, despite evidence that many curricula have been modified and simplified in an attempt to reach more students [3]. A process needs to be established that streamlines the learning curve for programming without lowering expectations. This process should encourage the understanding that programming is a tool used in computer science, rather than perpetuating the myth that programming and computer science are one and the same [4].

A. Breadth-First, Depth-First, Breadth-Also Approaches

Breadth-first approaches limit the exposure to programming languages and support concentration on basic concepts for future learning. One suggestion for a breadth-first introduction is to consider it an introduction or orientation to the subject area. Cook suggests that problem-solving skills should be given the largest portion (40%) of the course time, but other items, such as teamwork, computing tools, and career option information should be part of the course [5]. Others who have outlined breadth-first programs include those that are more oriented to discrete structures [6], [7] and those who tend to support a software engineering approach [8].

The depth-first approach is commonly a programming-first approach, though it would not preclude the incorporation of problem solving, algorithms, pseudocode, and diagramming as suggested by this study. The language is considered the focal point for problem-solving skill development with this model. Proponents for the depth-first model may go so far as to argue that programming-first should occur and be followed by a breadth-based course [9].

A carefully presented and justified breadth-also approach is outlined by Powers [10]. This curriculum is based on the pedagogical concepts of spiraling (referring to topic areas multiple times at different levels of difficulty and different perspectives throughout the curriculum), spacing (the concept of learners needing to see things over time to retain learning), and constructivist theories (understanding coming from what one has experienced). While this approach is supported by the Association for Computing Machinery (ACM)/IEEE Computer Society Joint Curriculum Task Force Computing Curriculum [1] guidelines and is carefully backed by pedagogically sound choices, research does not currently exist that illustrates increased learning success by participants.

B. *Programming Learning*

Problem-solving methods play a major role in most computer science topics. Strong foundations in domain recognition, pattern matching, problem organization techniques, and solution testing are important for later success in the discipline [1]. Early collections of heuristics for mathematical problem solving can be applied to programming, and one of the earliest sets of heuristics can be mapped to programming-related problem solving [11]. One of the first transfers of problem-solving techniques to programming is still applicable with today's languages and theories [12].

Shin, Jonassen, and McGee determined that problem solving for well-structured versus ill-structured problems require different skills [13]. Both require domain understanding and justification skills, but ill-structured problems require "meta-skills," such as planning and task decomposition. Larger systems design and analyses tend to be ill-structured; thus, organizational system problem-solving techniques are useful in software engineering environments [14], [15]. For early programming learning, a focus on well-structured problems helps to eliminate some of the learning curve required by poorly structured problems.

Computer science is, by nature, a collaborative discipline, often requiring team development and work [5]. Therefore, practitioners need to be made aware of team problem-solving techniques [10]. In fact, the techniques and processes found in mathematics, engineering, and scientific inquiry were built with the purpose of providing a tool to communicate problems, methods, and solutions. Methods such as diagramming and pseudocode are used to provide some format for problem solving that is not tied to a particular language. However, these tools provide a framework for future programming learning and have direct application within the profession. Thus, promoting these tools as part of a collaborative problem-solving toolset is a sensible approach.

Unfortunately, the transfer of a problem-solving technique to application in programming is a difficult step [16]. Thus, appropriate learning tools and frameworks from which new learners in computer science can build must be provided in order to reach a broader group of individuals. Making an assumption that preexisting problem-solving aptitude is sufficient for programming success does a disservice to all but those learners with exceptional self-motivation, intelligence, or experience. Similarly, those who use programming as the tool to introduce problem-solving techniques make the process more difficult by forcing multiple learning stresses on the learner [16]. By providing intermediate steps between general problem-solving techniques and programming, the transfer of problem-solving skill to programming language implementation will be more successful.

C. *Related Research*

Characteristics commonly linked to success in computer science include comfort with mathematics, computing, and problem solving. Byrne and Lyons show a tendency for persons with prior mathematical and science skill ability to have an easier time with learning programming [17]. Goold and Rimmer followed cohorts of students through the introductory classes and found that problem-solving skills are important for programming success. Further, they found that dislike of programming was an indicator for success or failure in the introductory sequence [18].

Other studies attempt to build self-efficacy measuring sticks for varying classes, tasks, and problems. This method seems to be a reliable method for obtaining sets of predictors for success. For example, a self-efficacy scale was attempted for programming success that appears to have reasonable measurements for reliability and validity [19]. Similarly, a recent study developed a scale for CS0 courses and found that prior problem-solving success, along with other characteristics, tended to play a role in CS0 course success [20].

Studies specific to approaches taken in introductory computer science education tend to be used to measure levels of success for favored approaches. For example, Bouvier used data collected with regard to student satisfaction, retention, and grades to determine the effectiveness of common learning experiences in a depth-first introductory environment [21]. While not well controlled and subject to numerous external variables, this study reported that retention and satisfaction increased while the grade distribution did not improve. Lewandowski and Morehead used a similar study approach and reported positive impacts on learning in their breadth-first environment [22].

Research also clearly shows that learners need to be encouraged to try problem-solving approaches that do not naturally occur to them. For example, new learners tend to spend very little time analyzing a problem and rely almost exclusively on trial and error [23]. This approach is quite contrary to experts who use a combination of associations, intuitions, and testing to solve a problem [24]. Clearly, less experienced problem solvers could use direction so that they can begin using a broader set of tools.

A large body of knowledge exists that clearly shows that working from within some sort of advanced structure tends to aid learners who enter a new domain of knowledge. Research in this area can be traced back to Dewey, who believed that learning was a series of reconstructions of knowledge [25]. Ausubel applied these ideas to language learning. He found that, to learn a language, one first perceives how the language works or applies and then subsumes that knowledge using his or her own experience for a frame of reference [26]. Mayer found that, in general, advanced organizers (tools that help provide a frame of reference for a new learner) do improve learning success [27]. In addition, Perfetti confirmed that advanced organizers tended to increase retention of knowledge [28]. Other works give direct support to the contention that advanced organizers support programming learning [29]–[31].

III. Method

The experimental design for this project supported an exploratory piece of research for which a hypothesis, and its corresponding null hypothesis, was tested. This hypothesis suggested a link between the implemented curricular changes in CS0 to increased programming success in the CS1 course. This study was a self-controlled one where participants were not randomly assigned to groups and where the control and treatment groups were not measured concurrently. This study was longitudinal in nature with the progress of participants being monitored for a term via data collection tools and the contact with each group lasting through a school year (two academic terms). Descriptive research techniques were used in the demographic and exit survey data collection tools to provide evidence as to the validity and reliability of data collected in this project.

This study followed a treatment/posttest/posttest design that followed two cohorts through their progress during attendance in CS0 and CS1 courses at the subject postsecondary school. For the purpose of this paper, the first posttest is referenced as a pretest since it was implemented at the beginning of the CS1 course, despite its occurrence after the treatment or control CS0 course. This design was derived from the base structure for nonequivalent control group designs provided by Cook and Campbell [32]. According to their definitions, this study should be regarded as a weak treatment versus strong treatment structure, since both sets of individuals did receive some instruction regarding algorithms and problem solving. Again, for the purposes of this study, the groups were referenced as control and treatment groups for clarity and consistency.

In an effort to address external variables, the study used data triangulation by collecting data on achievement, self-comfort ratings with related tasks, course satisfaction data, pertinent demographic data, and qualitative observational data from both achievement tests. Data points were selected with the knowledge of how similar data points had been collected by prior research (Section II). All data was collected using four data collection instruments: an entrance survey, a problem-solving pretest, an exit survey, and a programming posttest. Analysis triangulation was accomplished by combining traditional paired analysis statistical methods, data mining methods, and qualitative analysis approaches.

No control was exercised over the selection of members of the treatment or control group since each group consisted of all members of the courses for a given term. No advertisement or notice was given to prospective enrollees that courses differed, and no concurrent differences were present in CS0 offerings. The control group had completed the CS0 course in an academic term prior to the treatment group. Thus, each group represented an entire class of learners moving through the computer science curriculum for a given point in time. A typical cohort design was followed, with the exception that members of the control group could attend the CS1 course with treatment group members if they delayed continuation of their studies by a term. Persons fitting this description were not included in the study. In order to determine if the samples were representative of the sample population, demographics were collected and compared with school and discipline norms.

IV. Curricular Modifications

The set of concepts and techniques integrated into the preprogramming course were intended to introduce learners to the basics of algorithm development and problem solving. The concepts outlined in Section II of this paper were carefully considered in the selection of material and development of the modified course, with attempts to implement spiraling and spacing, and a strong focus on advanced organizers. The researcher selected these topics as items that were anticipated to have a direct impact on future learning in computer science and, in particular, programming learning. Treatments of diagramming, pseudocode, team problem solving, algorithms, test plans, documentation, requirements clarification, control structures, variables, and modularity were included in the preprogramming course. The control group had exposure to pseudocode, algorithms, modularity, control structures, and variables. The existing topics were extended or refocused to include the other areas of concentration (diagramming, algorithm testing, team development, requirements clarification, and documentation) for this course.

V. Analysis

The pretest and posttest score data points were the focus of the quantitative analysis, with secondary importance given to CS0 satisfaction ratings. In addition to these data points, qualitative analysis was undertaken on corresponding open-ended questions in the survey data collection tools and on observations of problem-solving and programming approaches in the testing tools.

The posttest scores collected by this study measured the ability of subjects to perform coding tasks. The pretest provided a baseline ability marking in problem-solving skills that was intended to take into account the natural variance of ability between individuals. The pretest and posttest were the same for both the control and treatment groups. However, the pretest and posttest were, by necessity, different. One could not measure programming skills before participants had any knowledge of them. Instead, efforts were made to create questions on both the pretest and posttest that followed a similar pattern of problem type and form.

The posttest and pretest data were analyzed using an analysis of variance (ANOVA) with a constructed independent variable that identified membership in the control or treatment group. An F-ratio was computed, along with its corresponding p-value to determine if the observed difference was statistically significant. This ANOVA served as the simple effects test for the repeated measure of the two tests across two groups: the control and the treatment group.

The data in Table I does not quite indicate that the posttest scores showed a statistically significant difference. The p-value of 0.089 was not within a confidence interval of 0.05. In addition, the relatively small sample size (control = 24 persons; treatment = 15 persons) places significant doubt as to whether the observed difference is an artifact of chance or a valuable observation. However, when one also considers the high p-value of 0.924 for the pretest, the case is strengthened somewhat since this number indicates that both sets of individuals started with a

TABLE I
ANOVA SIMPLE EFFECTS ON TEST SCORES ACROSS GROUPS

Raw Score From	Mean-square	F-ratio	P
Pretest	0.052	0.009	0.924
Post-test	42.339	3.050	0.089

similar raw set of skills. Again, however, these numbers cannot be construed as being significant.

Observation of patterns in the pretest and posttest provided additional information that supported the hypothesis. In the pretest, the use of pseudocode and pictures to supplement problem-solving occurred much more often in the treatment group than the control group. In fact, the quality of use of these helpers was consistently better in the treatment group. This finding indicated that learners in the treatment group were more prepared to use the advanced organizers than those in the control group, despite some exposure to them in both control and treatment CS0 courses. More significantly, in the posttest, ten instances were found in the treatment group where the participant used additional pseudocode or diagramming to aid in a coding solution, as compared with only four cases in the larger, control group. This observation supports the contention that command of these advanced organizers (pseudocode, diagramming, and algorithm testing) gave persons with similar raw problem-solving ability an additional tool to aid in programming tasks.

Satisfaction data was collected for the CS0 course at the beginning and end of the CS1 course. Participants ranked their satisfaction with the CS0 course on a scale of one to five, with five being the highest satisfaction level. The treatment group consistently ranked their satisfaction with the CS0 course higher than the control group. Comparisons of rankings using a traditional chi-square analysis found that course rankings differed significantly with a p-value of 0.002. A significant difference in satisfaction existed, which favored the treatment course.

Finally, open-ended questions in the survey instruments collected data that clarified and expanded on the statistically based data. Both groups expressed concerns at the beginning of the CS1 course regarding their ability to recall all of the pertinent programming language syntax necessary for the course. In other words, participants felt uncertainty with respect to how well programming would go for them. This uncertainty confirms the perception of need to produce helpers that aid learners in flattening their personal learning curves. The solution is to identify an appropriate set of advanced organizers to help a broad set of learners.

Members of both groups identified sections on problem solving, algorithms, and pseudocode as being the most important parts of the CS0 experience. These preferences indicate that students believed that these advanced helpers were useful in programming learning. In fact, members of the control group asked for increased exposure to these areas (and decreased exposure in a variety of other areas). Both groups increased their support for these areas in the exit survey beyond already high levels in the entrance survey. In addition, these areas of concentration were not mentioned as being the least useful parts of the CS0 course in either group. The treatment group exhibited full support for this topic area, with all 15 respondents listing algorithms, pseudocode, and problem solving as the most important part of CS0 in their exit survey. The control group was not as certain, with 17 of 24 responding in this fashion. However, participants found the link between these tools and programming to be present.

VI. CONCLUSION

This study produced a number of interesting results that provide qualified support for the contention that teaching problem-solving techniques and algorithm development prior to programming is beneficial for learning. Posttest programming scores for like tests differed, while pretest baseline scores did not, with the treatment group performing better in the posttest than the control group. Members of the treatment group tended to report a better experience in the CS0 course and saw more applicability for that course than members of the control group. In addition, treatment group members utilized pseudocode more consistently and applied the idea of drawing pictures to aid in writing code during their posttest experience. The combined results indicate that learners, in general, found that the CS0 course had a clear purpose and that they could successfully apply processes learned there as part of a programming strategy. The data suggests that this strategy led learners to greater success in learning their first programming language.

These results do not directly measure the effectiveness of breadth-first approaches, nor was the study intended to identify which of the three typical curricular organizations is most effective. In fact, any of the three approaches could successfully introduce problem solving, algorithm development, testing, pseudocode and diagramming prior to actual programming. However, these results may or may not be scalable to programs of different sizes or structures because the number of external variables makes the isolation of the curriculum treatment from other factors very difficult. In addition, the personal nature of learning does not indicate that one could, or should, apply these results with the belief that all learners would benefit from these particular organizers provided in this fashion. However, the results show sufficient consistency with existing research in pedagogy, computer science education, and programming learning to encourage educators to apply these tools as advanced organizers to programming learning.

REFERENCES

[1] *ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula*, 2001.

[2] M. Clancy, J. Stasko, and M. Gudzial, "Models and areas for computer science education research," *Comput. Sci. Educ.*, vol. 11, no. 4, pp. 323–340, 2001.

[3] A. B. Tucker, C. F. Kelemen, and K. B. Bruce, "Our curriculum has become math-phobic," in *Proc. Special Interest Group–Computer Science Education (SIGCSE)*, 2001, pp. 243–247.

[4] D. Powers and K. Powers, "Constructivist implications of preconceptions in computing," presented at the Information Systems Education Conf. (ISECON) 2000, Philadelphia, PA, 2000.

[5] C. Cook, "CS0: Computer science orientation course," in *Proc. Special Interest Group–Computer Science Education (SIGCSE)*, 1997, pp. 87–91.

[6] A. B. Tucker *et al.*, "Developing the breadth-first introductory curriculum: Results of a three-year experiment," *Comput. Sci. Educ.*, vol. 8, no. 1, pp. 27–55, 1998.
[7] A. Tucker and D. Garnick, "A breadth-first introductory curriculum in computer science," *Comput. Sci. Educ.*, vol. 3, pp. 272–295, 1991.
[8] D. Bagert, W. M. Marcy, and B. A. Calloni, "A successful five-year experiment with a breadth-first introductory course," *SIGCSE Bull.*, vol. 27, no. 1, pp. 116–120, 1995.
[9] C. Gray and M. Frazier, "Introducing computer science after programming," *J. Comput. Sci. Colleges*, vol. 18, no. 1, pp. 65–76, 2002.
[10] K. Powers, "Breadth-also: A rationale and implementation," in *Proc. Special Interest Group–Computer Science Education (SIGCSE)* , 2003, pp. 243–247.
[11] G. Polya, *How to Solve It?*. Princeton, NJ: Princeton Univ. Press, 1973.
[12] W. Mitchell, *Prelude to Programming*. New York: Reston, 1984.
[13] N. Shin, D. Jonassen, and S. McGee, "Predictors of well-structured and ill-structured problem solving in an astronomy simulation," *J. Res. Sci. Teaching*, vol. 40, no. 1, pp. 6–33, 2003.
[14] R. Ackoff, *The Art of Problem Solving*. New York: Wiley, 1987.
[15] S. Krantz, *Techniques of Problem Solving*. Providence, RI: Amer. Mathematical Soc., 1991.
[16] D. Woods, "An evidence-based strategy for problem solving," *J. Eng. Educ.*, p. 443, Oct. 2000.
[17] P. Byrne and G. Lyons, "The effect of student attributes on success in programming," in *Proc. Innovation Technology Computer Science Education (ITiCSE)*, 2001, pp. 49–52.
[18] A. Goold and R. Rimmer, "Factors affecting performance in first year programming," *SIGCSE Bull.*, vol. 32, pp. 39–43, 2000.
[19] V. Ramalingam and S. Wiedenbeck, "Development and validation of scores on a computer programming self-efficacy scale and group analysis of novice programmer self-efficacy," *J. Educ. Comput. Res.*, vol. 19, no. 4, pp. 367–381, 1998.
[20] A. Quade, "Development and validation of a computer science self-efficacy scale for CS0 courses and the group analysis of CS0 student self-efficacy," presented at the Innovation Technology Computer Science Education (ITiCSE), 2003.
[21] D. Bouvier, "Pilot study: Living flowcharts in an introduction to programming course," in *Proc. Special Interest Group–Computer Science Education (SIGCSE) 2003*, Reno, NV, Feb. 2003, pp. 293–295.
[22] G. Lewandoski and A. Morehead, "Computer science through the eyes of dead monkeys," presented at the Special Interest Group–Computer Science Education (SIGCSE), 1998.
[23] A. Schoenfeld, *Mathematical Problem Solving*. New York: Academic, 1985.
[24] E. Fischbein, *Intuition in Science and Mathematics*. Amsterdam, The Netherlands: Reidel, 1987.
[25] J. Dewey, *Experience and Education*, reprinted ed. New York: MacMillan, 1963.
[26] D. Ausubel, *Educational Psychology: A Cognitive View*. New York: Holt, Rinehart and Winston, 1968.
[27] R. Mayer, "The psychology of how novices learn programming," *Comput. Surv.*, vol. 1, pp. 121–141, 1981.
[28] C. Perfetti, "Levels of language and levels of process," in *Levels of Processing in Human Memory*, L. Cermak and F. Craik, Eds. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1979, pp. 159–181.
[29] F. Bailie, "Improving the modularization ability of novice programmers," in *Proc. Special Interest Group–Computer Science Education (SIGCSE)*, 1991, pp. 277–282.
[30] R. W. Holt, D. A. Boehm-Davis, and A. C. Schultz, "Mental representation of programs for student and professional programmers," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Norwood, NJ, 1987.
[31] A. Applin, "Second language acquisition and CS1: Is $* == **$?," in *Proc. Special Interest Group–Computer Science Education (SIGCSE)*, 2001, pp. 174–178.
[32] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Boston, MA: Houghton Mifflin, 1979.

Rob Faux (M'03) received the B.A. degree in computer science and mathematics from Luther College, Decorah, IA, in 1988, the M.S. degree in computer science from Minnesota State University, Mankato, in 1998, and the Ph.D. degree in computer science education from Union Institute and University, Cincinnati, OH, in 2003.

He currently teaches at Wartburg College, Waverly, IA, and his research interests include computer science education, distance education methods, curriculum development, algorithm visualization, database development, and expert systems.