The Design and Implementation of a First Course in Computer Programming for Computing Majors, Non-Majors and Industry Professionals within a Liberal Education Framework

Ronald J. Harkins
Miami University
1601 University Blvd.
Hamilton, OH 45011
513-785-3137
harkinrj@muohio.edu

Abstract

With declining interest and enrollments in computer programming courses, it has been necessary to consolidate course offerings resulting in a particular class consisting of different learning objectives for its representative student constituencies. This paper details the design and implementation of a first course in computer programming with a liberal education focus, but populated by computing majors, non-majors, and working professionals. Careful attention must be given to the liberal education theme and the proper instructional methodologies in order to meet the learning objectives of these three distinct student groups within the same classroom. Additionally, pragmatic teaching maxims will be provided to help ensure success in offering not only this programming course, but also any liberal education computer information systems course populated by different student groups with different associated course expectations.

KEYWORDS: computer programming, liberal education in technology, CS0, non-majors, Pair Programming, Active Learning

1. INTRODUCTION

Universities continue to struggle to address declining enrollments in certain computing and technical disciplines. Some colleges continue to offer a wide variety of computing courses to meet student needs, but with very low enrollments in each. Indeed, smaller computing classes taught in computer classrooms in an active learning format have been shown to improve learning and enrollment retention, as well as student satisfaction (Boyer, 2007). However, for many schools, offering a variety of such small classes has become a financial burden. Instead, they offer a smaller number of classes, or even a single section of a particular course, hoping to maximize its enrollment. Consequently, a particular course section can be populated by students with very different course expectations. Furthermore, designing course materials and teaching the course can present a challenge to the instructor. The Computer Science and Systems Analysis department at Miami University offers a course entitled "Introduction to Computer Concepts and Programming" (CSA 163). This first course in computer programming with Visual BASIC is sometimes taken by computing majors who lack algorithm development and programming ability for the object-oriented programming course in JAVA (CS1). Some working professionals also enroll in the course to acquire Visual BASIC programming skills. Finally, most students enrolled in this course are non-majors who take it to fulfill a

liberal education requirement for their degree, under the university's Miami Plan for Liberal Education.

2. MIAMI PLAN FOR LIBERAL EDUCATION

The Miami Plan for Liberal Education, a significant revision of an earlier liberal education core requirement for graduates of Miami University, was adopted in 1988. The Plan requires students to take a number of courses (usually 3 – 9 semester hours) in each of five foundation course groups, followed by a 3-course thematic sequence to provide an in-depth study in an area outside of the student's major, and culminating with a liberal education capstone experience. CSA 163 is a Group V (Mathematics, Formal Reasoning, Technology) foundation course of the Plan. Non-majors, in particular, take CSA 163 to meet this Miami Plan foundation course requirement.

To have a course designated as a Miami Plan course, a formal application must be submitted by the department to the university's Liberal Education committee. The application must clearly demonstrate how the course will meet and incorporate defined liberal education principles into the course. These principles include critical thinking, understanding contexts, engaging with other learners, and reflecting and acting. Some might contend that a skills acquisition course, such as a computer programming course, is incompatible with such liberal education principles. However, the CSA department was very attentive to these principles in the design of CSA 163 by focusing on problem-solving and ensuring a natural integration of each of these principles into the course, which strengthened the objectives and outcomes of the resultant course. This is especially important for computing majors and/or working professionals who might enroll in the course not seeking any Miami Plan liberal education requirement fulfillment, but rather acquisition of problem solving/programming skills in Visual BASIC.

3. APPLYING LIBERAL EDUCATION PRINCIPLES TO A COMPUTER PROGRAMMING COURSE

Infusing liberal education principles into a skills acquisition course, such as computer programming, can be especially advantageous to non-majors. It can help dispel misconceptions about the art and skill of programming, and programmers as "geeks" who work in isolation. Non-majors themselves provide diversity to the programming course, and the liberal education principles make it easier for them to understand the broader context of computer programming in helping individuals work with computers to enrich their own professional lives as well as the larger society that is becoming increasingly technological (Allen, 1990;Anderson, 2003; Brady, 2004). The problem-solving and logical reasoning skills utilized in a first course in computer programming transfer to end-user programming skills, such as macro creation, spreadsheet formula/function derivation, and dynamic web applications…all important to non-majors. Furthermore, social persuasion and self-efficacy can increase for learners, especially non-majors, in a computer programming course by incorporating liberal education principles into the course (Wiedenbeck, 2005).

3.1 Critical Thinking Principle

Problem solving strategies employed in a traditional college mathematics course are essentially the same in a first course in computer programming. The primary difference is that the problem's solution is implemented on a machine using a computer language to direct the solution. Thus, the logical reasoning and critical thinking skills which are so vital to success in mathematics are likewise crucial to success in computer programming. Furthermore, courses that emphasize the development of problem-solving skills and logical reasoning support the objectives of curricula grounded in liberal education (Ellison, 1980). Clarity in problem definition, accuracy of proposed algorithms, and the relevance of both input data and output information, require significant critical thinking and analysis (Fagin, 2006). Norris and Jackson (1992) investigated the effects of a BASIC programming course on students' critical thinking and mental alertness and found significant improvement in students' critical thinking skills at the conclusion of the course.

Whereas critical thinking skills might be more apparent for the computing major or working professional in a first course in computer programming, the non-major/liberal education student might struggle with the critical and analytical thinking processes in computer programming. Small group exercises and pair programming (to be discussed later in this paper) can assist non-majors in improving their logical reasoning and critical thinking skills. Furthermore, connecting the problems to be solved to non-majors' areas of interest or anticipated careers, can also help them focus their critical thinking in a relevant context (Allen, 1990). Layman and Williams (2007) found that only 34% of programming projects in a beginning programming course had any practical or socially relevant context. Addressing socially relevant problems, some with ethical considerations, can motivate liberal education students to realize the importance of critical thinking in the design of efficient, practical, and reliable algorithms and solutions to important societal problems whose solution can be significantly improved and tested using a computer (Bosse, 2000). In a first course in computer programming, debugging activities and inspection and appraisal of alternative solutions and code for a problem, especially in a group discussion, are ideal mechanisms to focus on, and subsequently improve students' critical thinking skills.

3.2 Understanding Contexts Principle

Students in a first course in computer programming, such as CSA 163, also add to their knowledge base about the conceptual framework, achievements, and societal issues in computer technology. This is accomplished by students reading a secondary "computer concepts" textbook and associated newsprint and internet articles, and participating in small-group discussions on topics or issues drawn from these sources. While computing majors and working professionals might already know a significant amount of the technical hardware, software, and systems related topics, this knowledge is balanced by the non-major/liberal education students' perceptions and contributions in the cultural and societal issues related to technology.

3.3 Engaging With Other Learners Principle

Students learn from one another. Working with fellow students on problem solutions using a computer proves invaluable to their success, as well as their confidence and self-esteem. Informal hierarchies in a computing classroom, such as a "novice" group, a "some background" group, or an "expert" group can be blurred, or somewhat dissolved by incorporating partnership/small-group learning activities into a course. This also tends to diffuse a defensive climate that can occur when competitiveness, rather than cooperation permeates computer learning (Barker, 2002;Garvin, 2004). To this end, pair programming is utilized in many computing courses, including CSA 163, a first course in problem solving and programming with Visual BASIC. With pair programming, two students share a single computer to complete in-class programming lab activities. One student, designated as the "navigator," reads instructions, and reviews program code and actions completed the other partner, the "driver," who uses the keyboard and the mouse to interact with the shared computer. These roles are periodically reversed throughout the laboratory activity to allow each partner to experience each of these roles. Both driver and navigator are actively involved in reviewing their shared work, debugging their program, and recommending alternative, and hopefully more efficient and accurate solutions to the problem under consideration. "Mixed" partnerships (i.e. computing major/non-major/working professional) seem to work best, with non-majors providing "user considerations" to a solution, while computing majors provide additional technical expertise, when needed. However, it is important to ensure that both members of the partnership contribute to their mutual learning, and dominant or dogmatic behavior (especially by a computing major working with a less technically secure non-major) does not exist. If allowed, this can not only add to the frustration and feeling of inadequacy by the non-major, but can also result in unfair grading, with "weaker" students receiving high scores for work that was primarily completed by the "stronger" student of the pair (McDowell, 2006). Some educators employ a pair programming derivative

wherein the roles of the navigator and the driver are not as pronounced. Chong and Hurlbutt (2007) conducted a pair programming study that found the pairings to be more effective when the driver and navigator roles were not so distinct, but rather overlapping, with both partners taking on driver and navigator roles concurrently.

The benefits to implementing pair programming activities into a first course in computer programming or any active learning computer course are many, especially in a class populated by different constituencies, such as computing majors, non-majors, and working professionals. The dialog between partners in explaining a particular construct or algorithm is sometimes more effective in their learning than from a traditional textbook or lecture. Problem solving and programming become a joint venture, and a more sociable, enjoyable, and satisfying experience (Preston, 2006).

Computing majors are somewhat empowered in helping their partner, while also allowing the major to discover new information in response to their partner's questions or observations. Working professionals bring "on the job" anecdotal commentary and suggestions to the problem solving activity being jointly developed. In fact, industry professionals working in pairs have reported higher job satisfaction than those who work alone (Williams, 2000). Non-majors also feel more comfortable discussing a problem with a peer, than perhaps their instructor (Preston, 2006). This is particularly important, as comfort level in a computer science class was found to be the best predictor of success in an introductory computer science course (Cantwell, 2001). Another study found that students who programmed in pairs in an introductory computer programming course were more confident, had higher course completion and passing rates, and were more likely to continue in some computer-related major of study (Werner, 2004). Another pair programming study conducted in 2004 at the University of Auckland (NZ) found that a higher percentage of paired students passed a software design and construction course, compared to students who worked alone on their projects. The majority of students in this study also expressed a desire to use pair programming in their future computing courses (Mendes, 2006). It has also been shown that programs written in pairs were completed in a shorter time, were of higher quality, and received a higher grade than those written alone (Benaya, 2007).

A pragmatic detriment to utilizing pair programming in an active learning, computing course occurs when the paired activities cannot be completed within the designated class period. Finding time to complete the project jointly, due to incompatible work and "after class" schedules can pose a significant hardship for students, especially returning/working students (VanDeGrift, 2004). Additionally, if instead of completing the work jointly outside of class, it is to be completed in class as a pair during the "next class" meeting, problems can arise when a member of a partnership fails to attend this subsequent class session. To minimize these scheduling problems, in light of the countless benefits to pair programming cited previously, CSA 163 utilizes pair programming only in completing shorter (30 – 45 minutes), directive lab activities, leaving more comprehensive programming assignments to be completed individually outside of class.

Doing this, also helps prevent one member of the partnership from becoming too dependent on the other partner in learning how to problem solve and program in Visual BASIC. This lack of independent thinking and action can be a detriment in completing current course exams independently, or even later on in a computing career, when certain actions, technical decisions, or solutions must be derived on one's own. On the other hand, working with a partner can be valuable to "team programming," which occurs widely in industry. In fact, a final team programming project is recommended in CSA 163, with enough "lead time" provided for team members to arrange work schedules accordingly. In most cases, the "team" becomes simply the "pair" from the pair programming course lab activities, with perhaps one or two additional members, as the social/working connection that was so helpful throughout the semester in pair programming is continued and strengthened by this final team programming project.

3.4 Reflecting and Acting Principle

Thinking critically and understanding contexts for knowledge in an active learning environment naturally lead to reflection and informed action. Students in a first course in computer programming, such as CSA 163, have ample opportunities to reflect and act on problem solving methodologies, and the subsequent implementation by their computer program. Pair programming laboratory activities invite the students to alter code and report on the impact of these modifications. When testing programs, students are encouraged to use data from various data sets (e.g. integral, real, character, or string) or from various data ranges (e.g. above 500, between 100 and 500, and below 100) and report on the accuracy and relevance of the solution output. Students are asked to provide data ranges for input data that conform to "real life" and investigate the accuracy of related output information. In CSA 163, utilizing the Visual BASIC IDE, students must be aware of user (customer) requirements, and reflect on their program's usability accordingly. Working programs must be user-friendly, and "forgiving" to users, when they err in interacting with the program. In the pair programming lab activities, one student assumes the role of the "user/customer," while the other acts as the "programmer" in implementing changes to the code or interface in response to the user's concerns and suggestions.

When real-life problems (e.g. population growth in underdeveloped countries, mortality rates in Darfur, computer recycling and distribution) are studied in the liberal education CSA 163 course, students are asked to reflect, in writing, on the output generated by their computer program. Indeed, written communication is a critical component of any liberal arts curriculum. The architects of a liberal arts curriculum who integrate it with oral and written communication requirements receive high praise and support from industry leaders who find their employees deficient in vital communication skills needed both within internal departments of a company, as well as among units operating around the world. For educators, as well as students, incorporating meaningful writing components into computing or technology-driven courses can be a difficult and time consuming, and sometimes perceived as "forced" by students, with writing assessment responsibilities and guidelines both vague and undefined (Kaczmarczk, 2004). All three representative student groups in a typical CSA 163 class (computing majors, non-majors, and working professionals) might question the value and need of writing activities woven throughout the course. Curriculum developers and instructors in technology courses must work hard to make such writing requirements meaningful and clearly connected to the technical content of the course. Walker (1998) identifies some activities in a computing course that could have a writing component. These include explaining why something happens in a program, comparing two approaches or algorithms, justifying one's answer, or discussing the purpose of a procedure or code block. He further requires students to document programs heavily and meaningfully, and returns undocumented programs to students ungraded. Dugan and Polanski (2006) provide advice to computing course instructors wishing to incorporate writing activities into their courses. This advice includes giving writing assignments a real world context, demonstrating the importance of writing in computing-related courses, requiring revision of writing submissions by students, and conducting peer reviews of writing assignments. Ladd (2003) suggests reducing the number of programming assignments significantly, and instead, having two due dates for each assignment. The first deadline is for the initial submission, while the second date is for the submission of a revised program incorporating modifications suggested by the instructor, as well as a one page narrative detailing how these changes addressed the instructor's initial evaluative comments. Anewalt (2002) acknowledges that integrating writing into a computing course for the first time can be both intimidating and challenging for the instructor. She contends that the key to a successful writing experience for students requires the instructor to clearly connect such writing with the course objectives, making expectations clear to the students, and to keep the grading of the written components both consistent and simple.

In CSA 163, short answer questions, such as "Explain the differences among the numeric data types for variables in Visual BASIC." or "What advantages do you see for event-driven programming for both the programmer, and the end-user?" are included on every examination. Furthermore, extensive and meaningful documentation is required for all submitted programs, as well as code segments of the lab activities written by a programming pair. In addition, two research/opinion papers are included in CSA 163, one of which involves taking a previously written program and having someone with very little computing experience run it. In this reflection paper, the CSA 163 student writes a short summary report of the user's reactions, suggestions, and even frustrations with the original program, and subsequent action(s) taken by the CSA 163 student-programmer to accommodate, or reject the user's comments, recommendations, or complaints. For some students in CSA 163, especially the computing majors, this is an eye opening and somewhat humiliating experience, as they tend to be very protective, even defensive, of their written code, and rather unresponsive to criticism of it, especially from someone knowing little about computers. On the other hand, the non-majors enrolled in the same CSA 163 class are more receptive to non-technical user's concerns, as they can better relate to someone without a high level of technical programming background or ability. Working professionals enrolled in the same CSA 163 class are accustomed to meeting customer needs and requests in their daily work, so making software user-friendly is both obvious and apparent to them. Written communication is an ideal and necessary tenet of any liberal education technical course and an excellent vehicle for utilizing the 'reflecting and acting' liberal education principle of the Miami Plan for Liberal Education in CSA 163.

Integrating the four principles of the Miami Plan for Liberal Education (critical thinking, understanding contexts, engaging with other learners, and reflecting and acting) into CSA 163, a first course in problem solving and computer programming, enriches the course, and makes it a more satisfying and meaningful experience for all three student constituencies (computing majors, non-majors, and working professionals) who regularly enroll in this course. The liberal education principled model described in this paper can likewise be used in developing and delivering similar computing and technical courses in information systems, information technology, and business technology, as well as other computing-related fields and disciplines.

4. PEDAGOGICAL ADVICE

The author has taught this first course in computer programming (CSA 163) every semester since it was offered as a Liberal Education foundation course at Miami University in 1988. In the early years, when enrollments were high, several sections of the course were offered, with "day" sections primarily comprised of traditional age college students fulfilling their Miami Plan Group V liberal education requirement, or beginning a major in computer science. Working professionals and non-traditional returning students enrolled primarily in "evening" sections of the course. In general, teaching methods and course materials could be developed in alignment with the learning styles and cognitive behaviors of the types of students in a particular section. Discovery learning challenges can be woven into activities/lectures/demonstrations for computing majors. Reflective activities (e.g. written opinion positions, small group discussions) are especially valued by the liberal education students. Finally, busy, working professionals appreciate teaching/learning activities with real-life impact that produce tangible, useful results/skills that clearly connect to their responsibilities in the workplace.

As the years passed and enrollments and interest in computing courses and associated careers declined, so too did the number of sections of this first course in computer programming. Consequently, fewer sections of CSA 163 were offered and were populated by all three types of students (liberal education students, computing majors, and working professionals). Different instructional techniques had to be used to meet the needs of all three of these types of students, and their corresponding learning styles in the same classroom. While this was challenging, it was not impossible. Furthermore, the resulting student diversity improved the course by providing alternative

viewpoints, questions, and discussions from each of these constituencies.

The author provides the following set of pedagogical maxims to assist instructors in offering a liberal education technology course, such as CSA 163, or any "first course" in computer science, computer technology, information systems, or business technology to a class with varied interests, needs, learning styles, and reasons for taking the course.

4.1 Just Do It.

Incorporate online, active learning into every class session. Try interrupting lectures and demonstrations with online active learning opportunities for the students. Include both practice/mastery and discovery learning in these online activities to provide needed information, while encouraging intellectual curiosity in the students.

4.2 Mix It Up.

Try to avoid class sessions that are exclusively lecture or exclusively laboratory. Adding variety to classroom activities will increase student interest and participation. Try to include lecture segments enhanced by short laboratory activities and demonstrations that solicit student feedback, modification, debugging, or completion. Short quizzes and group discussions can also be added to the mix.

4.3 Can I Help You?

Try employing the pair programming paradigm, described earlier in this paper. Students "talk the talk" and can explain some things better than you! "Mixed partnerships," consisting of computing majors, non-major liberal education students, and working professionals encourage different perspectives in their problem solving and computing experiences. Become a "helicopter instructor," moving from pair to pair, acting as a facilitator as you hover, especially when noticing that little interaction is occurring between the partners, or one member seems to be doing all the work. Engaging with other learners is an important component and principle of liberal education.

4.4 Put It In Writing.

Try to encourage written communication throughout the course activities, and not simply in one or two isolated writing projects. The latter might simply be dismissed by the students as simply another course requirement that must be tolerated and completed for a grade. Connecting and including short writing experiences into pair programming lab activities, program/computing assignments, quizzes, and examinations will help reinforce the value and importance of written communication in computer study, and subsequent computer-related careers. It also allows them to employ the Reflecting and Acting principle of liberal education in these writing activities, especially on opinion/reflection statements and papers. If opting to include a significant research/opinion paper, try to connect it to the educational objectives and needs of the students in the class. Providing topical choices can make writing activities meaningful for each of the different student constituencies in the course. For example, consider a persuasive essay on ethical behavior involving technology, for liberal arts students (Cliburn, 2006); investigating a programming feature or topic not covered in the course, and evaluating its usefulness, for computing majors; or summarizing and resolving a technical crisis at work, for working professional students.

4.5 Get Real.

Try to incorporate contemporary, real-life examples in lectures. lab activities and programming/computing assignments. Ask working professional students to provide a real-life application of a class activity or programming/computing assignment (e.g. inventory management, distribution models, promotion/reward mechanisms). Incorporate financial applications with business practices that are characteristic of the day-to-day life of a student. Connect a programming construct (e.g. parameters of functions) to application software they are familiar with (e.g. EXCEL functions), or even to real life activities, such as sports (e.g. passing and receiving in football, to parameter passing in programming). This can illustrate and employ the Understanding Context principle of liberal education defined earlier in this paper, especially in the abstract realm of problem solving.

Furthermore, try to include problems that involve social or ethical dimensions (e.g.

population growth, health appraisals, identity theft statistics, homelessness data, etc.) in keeping with the Reflection and Action principle of a solid liberal education course in any discipline.

4.6 But Does It (Always) Work?

Encouraging the development of robust and reliable algorithms in problem solving can be accomplished by requiring extensive testing of solutions implemented by a computer program or application. Extensive testing also increases the confidence of the student-programmers in the overall reliability and accuracy of their work (Edwards, 2003). Consider having one member of a partnership in a pair programming activity try to "break" a program segment developed by the "other member" of the pair using invalid input data. Can the "break" be fixed at this point in the course? Perhaps not. Can they "discover" a solution, on their own? Are program results accurate, possible and realistic when applied to everyday life? Answering these important questions requires the students to apply both the critical thinking, and the reflecting/acting principles of liberal education

Finally, consider the "test first" programming strategy popularized by extreme programming (Edwards, 2003), which would be especially interesting to computing majors in the course.

4.7 How's It Going?

Try to evaluate student progress frequently and provide quick turnaround time and meaningful feedback on evaluative measures. Maintaining a web-based dynamic grade book that informs students of their current average for any given day or week of the semester or quarter can be useful, informative, and motivating for students. Incorporate variety in evaluation (e.g. written assignments, online programming assignments, hourly tests, short quizzes, lab activities, position/research papers, etc.). Also, try to provide formative evaluation opportunities such as short online topical practice tests. Some evaluation can be particularly focused toward a particular constituency in the class. For example, liberal arts students might appreciate a question on the intended customer, the inherent value, or the user-friendliness of a particular program; whereas a computing major might be motivated by an open-ended discovery learning challenge, such as "Do you think it is possible for your program to…?" Consider peer review of program assignments by electronically delivering a student's program to individual workstation monitors or a wall-mounted classroom projection screen soliciting both commendations and constructive criticism from the students. Try to review a different student's work each time this peer review process is conducted. Peer review can also be conducted in a more personal and informal manner between partners in a pair programming activity, but this usually occurs naturally, without any instructor initiation.

4.8 Be There.

Finally, plan to provide reasons for students to attend class, other than simply giving points for attendance. This can result in passive, even bored, attendees. Incorporate an "event" (e.g. quiz, demonstration, video clip, lab activity, etc.) into every class meeting so students see a real value and purpose of attending every class session of your computing course. When posting lecture notes on the web, consider making them intentionally incomplete (i.e. more like an outline) that will be completed by them in class. Posting complete, detailed lecture notes, on the other hand, without providing additional in-class activities, might encourage students to skip the class, finding something more important and meaningful in their busy lives to attend to. Finally, remember that students in a computing class are "active learners," so try to identify some kind of online activity to include in every class session.

5. CONCLUSION

Teaching a first course in computer programming or in any computing/information systems area in a liberal education framework can be a challenge, especially if the students in the same classroom have different needs or objectives for enrolling in the course, such as liberal education students, beginning computer science/technology majors, or working professionals seeking to acquire technical skills for their current job. A recent panel of computer science educators (Walker, 2003) held that a computer science curriculum in a liberal arts environment

should contain a firm foundation in technical computer science, a commitment to problem solving, integration of the social impact and ethical issues related to computing, and development or oral and written communication skills, among others.

The relevancy of this list can be extended beyond computer science to any computing field (e.g. information systems, computer technology, business technology) delivered and studied in a liberal education environment. Some might think that skills acquisition courses, such as a first course in computer programming, and a liberal education courses, grounded in observation, reflection, and communication, are mutually exclusive. This isn't necessarily true. A technology-driven course, while focused on problem solving and skill acquisition, can nonetheless be structured to incorporate the critical thinking, understanding contexts, engaging with other learners, and the reflection/action principles described in this paper to produce a course rich in both liberal education and skill acquisition. This mix of technical skills and liberal education principles is very appropriate for today's students and tomorrow's careers in an increasingly technical and culturally diverse society.

6. REFERENCES

Allen, J., H. Porter, T. Nanney, and, K. Abernethy, (1990) "Reexamining the Introductory Computer Science Course in Liberal Arts Institutions". Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education, pp. 100 – 104.

Anderson, P., J. Bennedsen, , S. Brandorff, , M. Caspersen, and J. Mosegaard (2003) "Teaching Programming to Liberal Arts Students: A Narrative Media Approach," Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science, pp. 109 – 113.

Anewalt, K., (2002) "Experiences Teaching Writing in a Computer Science Course for the First Time," Journal of Computing Sciences in Colleges, Vol. 18, pp. 346 – 355.

Barker, L., K. Garvin-Doxas, and, M. Jackson ,(2002) "Defensive Climate in the Computer Science Classroom". Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, pp. 43 – 47.

Benaya, T., and E. Zur, (2007) "Collaborative Programming projects in an Advanced CS Course", Journal of Computing Sciences in Colleges, Vol. 22, pp. 126 – 135.

Bosse, M., and N. Nandakumar (2000) "Real-World Problem-Solving, Pedagogy, and Efficient Programming Algorithms in Computer Education", ACM SIGCSE Bulletin, Vol. 32, pp. 66 – 69.

Boyer, K., R. Dwight, and C. Miller (2007) "A Case for Smaller Class Size with Integrated Lab for Introductory Computer Science", Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, pp. 341 – 345.

Brady, A., P. Cutter, and K. Schultz (2004) "Benefits of a CS0 Course in Liberal Arts Colleges", Journal of Computing Sciences in Colleges, Vol. 20, pp. 90 – 97.

Cantwell, B., and S. Shrock (2001) "Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors", Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, pp. 184 – 188.

Cliburn, C., (2006) "CS0 Course for the Liberal Arts", Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, pp. 77 – 81.

Chong, J., and T. Hurlbutt (2007) "The Social Dynamics of Pair Programming", Proceedings of the 29th International Conference on Software Engineering", pp. 354 – 363.

Dugan, R., and V. Polanski (2006) "Writing for Computer Science: A Taxonomy of Writing Tasks and General Advice", Journal of Computing Sciences in Colleges, Vol. 21, pp. 191 – 203.

Edwards, S. (2003) "Improving Student Performance by Evaluating How Well Students Test Their Own Programs", Journal on Educational Resources in Computing, Vol. 3, pp. 1 – 24.

Ellison, R., (1980) "A Programming Sequence for the Liberal Arts College",

Proceedings of the 11th SIGCSE Technical Symposium on Computer Science Education, pp. 161 – 164.

Fagin, B., J. Harper, and L. , Baird (2006) "Critical Thinking and Computer Science: Implicit and Explicit Connections", Journal for Computing Sciences in Colleges, Vol. 21, pp. 171-177.

Garvin, K., and L. Barker (2004) "Communication in Computer Science Classrooms: Understanding Defensive Climates as a Means of Creating Supportive Behaviors", Journal on Educational Resources in Computing, Vol. 4, pp. 1 – 18.

Kaczmarczk, L., G. Kruse, and, D. Lopez (2004) "Incorporating Writing into the CS Curriculum", Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, pp. 179 – 180.

Ladd, B. (2003) "It's All Writing: Experience Using Rewriting to Learn in Introductory Computer Science", Journal of Computing Sciences in Colleges, Vol. 18, pp. 57 – 64.

Layman, L., L. Williams, and K. Slaten (2007) "Note to Self: Make Assignments Meaningful," Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, pp. 459 – 463.

McDowell, C., , L. Werner, and H. Bullock (2006) "Pair Programming Improves Student Retention, Confidence, and Program Quality", Communications of the ACM, Vol. 49, pp. 90 – 95.

Mendes, E., L. Al-Fakhri, and A. Luxton-Reilly (2006) "A Replicated Experiment of Pair Programming in a 2nd Year Software Development and Design Computer Science Course", Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, pp. 108 – 112.

Norris, C., and, L. Jackson (1992) "The Effect of Computer Science Instruction on Critical Thinking Skills and Mental Alertness," Journal of Research on Computing in Education, Vol. 24, p. 329.

Preston, D. (2006) "Adapting Pair Programming Pedagogy for Use in Computer Literacy Courses", Journal of Computing Sciences in Colleges, Vol. 21, pp. 84 – 93.

VanDeGrift, T. (2004) "Coupling Pair Programming and Writing: Learning About Students' Perceptions and Processes," Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, pp. 2– 6.

Walker, H., M Jipping, and D., Baldwin (2003) "The Computer Science Major Within a Liberal Arts Environment", Journal of Computing Sciences in Colleges, Vol. 19, pp. 99 – 101.

Walker, H. (1998) "Writing within the Computer Science Curriculum," ACM SIGCSE Bulletin, Vol. 30, pp.24–25.

Werner, L., B. Hanks, and C. McDowell (2004) "Pair programming Helps Female Computer Science Students," ACM Journal of Educational Resources in Computing, Vol. 4, pp. 1 – 8.

Wiedenbeck, S. (2005) "Factors Affecting the Success of Non-Majors in Learning to Program,"Proceedings of the 2005 International Workshop on Computing Education Research, pp. 13 – 24.

Williams, L., R. Kessler,and, W. Cunningham (2000) "Strengthening the Case for Pair Programming,"IEEE Software.