2018 ASEE ANNUAL CONFERENCE & EXPOSITION
JUNE 24–27, 2018
SALT PALACE CONVENTION CENTER
SALT LAKE CITY, UT

Paper ID #23852

Enriching Communication in Introductory Computer Science Courses: A Retrospective of the Agile Communicators Project

Leo C. Ureel II, Michigan Technological University

Leo C. Ureel II is a Lecturer in Computer Science at Michigan Technological University. He has worked extensively in the field of educational software development. His research interests include intelligent learning environments, computer science education and software engineering. He currently has primary responsibility for the introductory programming courses at Michigan Tech.

Dr. Charles Wallace, Michigan Technological University

Dr. Charles Wallace studied Linguistics at the University of Pennsylvania and the University of California before earning his Ph.D. in Computer Science at the University of Michigan. He has been on the faculty of the Michigan Tech Computer Science Department since 2000. His experiences as a computer scientist, linguist, and software developer drive his research exploring how humans can better understand, build, and use software. His work has been funded by the National Science Foundation, Google, Microsoft Research, and the U.S. Department of Defense.

Dr. Wallace's Agile Communicators project, supported by an NSF IUSE award, seeks to build an enhanced curriculum for computing programs that emphasizes inquiry, critique and reflection, grounded in authentic software development settings. Tools in this project include process oriented guided inquiry learning, automated feedback to students through an intelligent tutoring system, case studies in software communication, and guided reflective exercises on team communication. As part of this research, the Agile Communicators team has investigated communication practices in a variety of student and professional software development environments.

Wallace has been intimately involved with undergraduate Computer Science curriculum development since his arrival in 2000. He cofounded Michigan Tech's Software Engineering degree program in 2003. Wallace currently serves as Director of Undergraduate Programs for the Computer Science Department. In conjunction with his research projects, he has founded local outreach efforts in computer education for middle and high school students and digital literacy for senior citizens.

Enriching communication in introductory computer science: A retrospective on the Agile Communicators project

Abstract

Among software professionals and educators, the quality of team communication is acknowledged as a key factor in the success or failure of software projects [1, 3]. Successful communication in the workplace is a process requiring more than technical mastery of standard genres. Developers must make strategic communication decisions, and they must be agile — flexible, proactive, and creative — in these decisions[9].

The *Agile Communicators* project seeks to promote productive, strategic communication among computing students. Our approach constitutes a cognitive apprenticeship [6] that engages students in authentic software settings and articulates processes that are traditionally left implicit. Over the two years of this exploratory project, we have enriched the communication environment in our early Computer Science courses, by engaging and encouraging students to articulate their questions, ideas, and concerns, and by enhancing the means by which instructors communicate with students. Our results to date indicate improvement in student performance, a clearer understanding of the place of communication in the lives of computing professionals, and an increase in persistence among students in our computing degrees.

Our products and interventions fall into two broad categories:

Engaging students as sophisticated communicators. Student practice and reflection on communication as part of the problem-solving process have been integrated into introductory programming courses (§1). Programming lab assignments include as deliverables not only the final result of functional code but also intermediate guided inquiry exercises where students articulate the steps of their analysis and design. Students complete lab assignments through pair programming and reflect on the progress of their communication skills with their partners over the sequence of assignments.

In a later team software project course, students consider more sophisticated communication scenarios (§2). Early in the team software course, we expose the students to real communication challenges that others have faced – the experiences of earlier student teams, and the experiences of professionals. Through a pattern approach, students inquire into the design choices of written and oral communication acts in real software projects. Later, during their project development, we ask them to reflect on the communication challenges they are facing, and to observe the communication choices that fellow teams have made.

Broadening the reach of instructor communication through computer mediation. A common principle of agile software approaches is constant feedback and reflection on individual and group progress. The feedback that instructors can provide face to face with students in lecture, lab, or office hours is precious but limited. We are inspired by constructivist approaches to computer mediated education [15, 8]. Within an explicitly social and communicative learning environment, digital tools can serve as much more than information repositories and rote graders. The key is in the design of interactive exercises that convey instructor intent and know-how through our tools. The emphasis in our interventions is on a three-way conversation between student, machine, and instructor.

1. Acknowledging and practicing communication in introductory programming

Instructors in introductory computing courses have a tremendous opportunity and responsibility as role models for good communication practices in the classroom and the lab. We emphasize the need to ask questions, provide explanations, and share ideas. Students are organized into pairs for the purpose of working on labs and assignments.

Pair programming has become a respected tradition in computer science, both in industry and academia [2, 16, 10]. In pair programming, teams of two work on the same design, code, or test. Sharing a computer,

one student is the "driver" who controls the keyboard and enters code. The other student is the "navigator" who constantly reviews the code in an effort to eliminate any defects. In our programming lab setting, we ask both partners to participate actively, talking aloud and narrating every thing they do and explaining every decision they make.

Our choice of problems encourages student collaboration and forces communication in a natural way. For example, as an early program that operates as an ice breaker, students program the computer to display characteristic information about their partner, such as name, hobbies, favorite color, etc. In another assignment, students are asked to work together to develop a one act play with two actors - each student is instructed to write one part but code the other. In a MadLibs-style assignment, students each create a story form, and select words to fill-in their partners blanks.

Interspersed between lab problems, students are given open-ended critical thinking problems and instructed to work together to resolve the questions. Their answers to these questions will then be used to solve later programming problems. Then at the end of the lab, we have students reflect critically on themselves and as pairs. Results from these self-reflections indicate that students feel more confidence and stronger in their abilities when working in pairs.

2. Team software course

Our Team Software Project course includes an introduction to the concept of software process, focusing on the Scrum framework [12]. Building on two years of experience with programming, software design and computer systems, students take on a semester-long project, with the instructor acting as client. The technical toolset developed in introductory courses is brought to bear on a real software problem. Here is where the notion of software process – the practice of creating software products in a replicable, reliable way – can be addressed and put into action. Techniques for effective communication are obviously an important component of this agenda.

One advantage of placing our instruction in this context is that Scrum explicitly acknowledges the importance of repeated, well-constructed communication. Many of the iconic practices of Scrum - stand-up meetings, sprint retrospectives, planning poker - are designed to increase discussion, reflection and debate, all of which help to strengthen the software process. The message that we wish to add is that Scrum, or any other process framework, can provide only broad guidelines for communication, not narrow, comprehensive rules; as professionals, they will be expected to create genres appropriate to the needs of the project [13]. For instance, team members may follow the practice of daily standup meetings, but it remains to their creative powers to determine what activities follow from the information shared at the standup.

We use a process of guided inquiry [5], where students construct their own interpretations of the subject matter through critical thinking and problem solving. This approach fits the topic well: the search for meaning within a given communication setting is complex, and different observers may see different patterns of communication in play. Guided inquiry allows students to take ownership of their interpretations; at the same time, we consciously steer students away from rote, simplistic answers that ignore the complexity of communication.

Survey results for the Team Software Project course strongly indicate that our instructional material is effective at building awareness among students of the importance of communication in real software development[9]. To a lesser degree, they indicate a moderate level of agreement that the activities helped to improve teams' communication process. Qualitative analysis of team communication activities through the course indicate an increase in sophistication of design, and explicit mention of communication design among students.

3. Blending Human and Computer-Mediated Code Critique

In the introductory programming courses, the WebTA tool provides tight instructor-tailored commentary on student code, essentially providing a virtual TA experience even when instructors are unavailable. WebTA not only reports on automated test results but also analyzes and critiques style and design, searching for positive patterns and negative antipatterns specified by the instructor.

Sending the student the error messages and warning generated by Java during compilation and runtime testing can be helpful, but provides little more that the students can obtain for themselves. The value an instructor brings to the development process is the ability to apply domain knowledge to the interpretation of such messages in relation to the student's code. As a pedagogical tool, WebTA allows the instructor to encode domain knowledge into rules which, when triggered by the student's code, warnings or error messages, produces the best feedback for the assignment.

For example, a common mistake in first year code is an off-by-one-error, where the program either stops one step short of its goal, or tries to go one step beyond. Knowing that students have read in twelve data elements for a given assignment, the instructor can create a rule looking for this common mistake and suggest that the student check the nearest loop for an off-by-one error.

The pattern used to trigger the rule does not have to come from Java generated messages. Often an instructor can anticipate common coding mistakes. For example, after covering iteration and moving on to modularity, we often see students adding empty for-loops in their methods. We call this Knee-Jerk code; a reflex action because students are still thinking about the previous topic covered in class and assume they must need to have it in their code. Here, the instructor can create a rule looking for a for-loop with no body. If found, this pattern triggers guidance suggesting that the student think about how each line of code contributes to the solution of the problem. If the code does not contribute, suggests the canned feedback, remove it.

Nor does the pattern have to be negative. The instructor can create a rule to identify a targeted code structure and pat the student on the back when they get it right. For example, when studying encapsulation, the first time a student creates a private instance variable with an accompanying accessor or mutator, a bit of on-the-spot praise can help engrain the habit.

Overall, students have indicated a high level of satisfaction with the use of WebTA. Students like receiving feedback in tight cycles of design, implementation, feedback, and reflection; especially when they are working at times that the instructor is unavailable. One drawback that we are investigating include the tendency of some students to use WebTA as a testing facility. Another subject of future work is trying to understand why a small number of students generate a high number of submissions (200+) before the assignment deadline.

4. Exploring Discrete Structures through Computer-Mediated Lab Experiences

Our introductory Computer Science course on discrete mathematics now involves lab exercises using the Alloy language and analyzer [7]. The logic at the heart of this tool combines the quantifiers of predicate logic with the operators of the relational calculus to provide a highly expressive specification language. An Alloy program asserts a set of constraints on mathematical structures over a program-specified signature. The Alloy Analyzer searches within programmer-specified bounds for models satisfying an Alloy program, through a SAT-based constraint solver.

The feedback provided by the Alloy Analyzer has the potential to eliminate common misconceptions among students. Compared to a traditional approach where students simply submit written answers to homework problems, students working on Alloy problems get immediate critique of the wellformedness and satisfiability of their responses. With a traditional pencil-and-paper exercise, students can remain in a false state of confidence, and deficiencies in understanding are not exposed until after the exercise is graded. Moreover, students in a lab setting are working in the pair programming configuration that they are familiar with

from their introductory programming courses; in this configuration, they are interacting not only with the computer and instructor but equally intensively with their partners.

Students can engage with Alloy at various levels of sophistication. For our early Discrete Structures course, we do not expect students to build sophisticated Alloy code from scratch. We design exercises carefully to take students from *observers* of program behavior to *tweakers* of search parameters and *builders* of more substantial constraints [14]. Alloy's expressive flexibility allows us to use it for a variety of topics. Constraints may be expressed in Alloy as first-order formulas, as relational expressions, or a combination of both. This flexibility gives us an opportunity as educators to illustrate a conceptual continuum in the course material. Starting with sets and relations, we state Alloy constraints as relational expressions. Later, when we cover first-order logic, we introduce this other style of expression in Alloy and show how it compares to the relational style. Finally, we address problems in graph theory using both styles, and taking advantage of the Alloy Analyzer's visualization functionality.

A study is being conducted comparing two versions of the discrete mathematics course, one with the programming labs and one without. Preliminary results indicate significantly higher performance on problems involving logic among students who participate in the labs.[4]. In addition, we have sought to determine factors that predict better performance using discrete structures. Our study indicates that success in discrete structures is reliably predicted by confidence, perceived usefulness and motivation. We are in the process of determining to what extent the use of the Alloy tool contributes to increases in these factors.

5. Future work

As we continue to develop our communication-oriented interventions, we plan to look more longitudinally on the effects of early communication-intensive instruction. In this regard, we are encouraged by the results of an initial study on student retention in our computing degrees, indicating an increase in persistence among our majors in staying within the computing disciplines [11]. Further study is needed to look at the communication practices and attitudes among our students as they venture into upper-level and graduate courses and into the workplace. Moreover, we wish to move our exploratory project into a broader sphere, involving other institutions.

References

[1] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser. SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering. *Computer*, 48:106–109, 2015.

[2] K. Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[3] P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2014.

[4] L. Brown, A. Feltz, and C. Wallace. Lab exercises for a discrete structures course: Exploring logic and relational algebra with alloy. In *23rd Annual Conference on Innovation and Technology in Computer Science Education*, 2018.

[5] J. Bruner. The act of discovery. *Harvard Educational Review*, 31(1):21–32, 1961.

[6] A. Collins. Cognitive apprenticeship. In R. Sawyer, editor, *Cambridge Handbook of the Learning Sciences*, pages 47–60. Cambridge University Press, 2006.

[7] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2012.

[8] D. Jonassen, M. Davidson, M. Collins, J. Campbell, and B. B. Haag. Constructivism and computer-mediated communication in distance education. *American Journal of Distance Education*, 9, 1995.

[9] S. Kumar and C. Wallace. Instruction in software project communication through guided inquiry and reflection. In *Frontiers in Education (FIE)*. IEEE, 2014.

[10] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception and persistence. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 602–607. IEEE, 2003.

[11] L. Ott, B. Bettin, and L. Ureel. The impact of placement in introductory computer science courses on student persistence in a computing major. In *23rd Annual Conference on Innovation and Technology in Computer Science Education*, 2018.

[12] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Prentice Hall, 2001.

[13] C. Spinuzzi. *Tracing Genres through Organizations: A Sociocultural Approach to Information Design*. MIT Press, 2003.

[14] L. Ureel and C. Wallace. Discrete mathematics for computing students: A programming oriented approach with alloy. In *Frontiers in Education*, 2016.

[15] L. S. Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

[16] L. Williams and R. Kessler. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002.