Agile Communicators: Cognitive Apprenticeship to Prepare Students for Communication-Intensive Software Development

Shreya Kumar
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
Email: shreyak@mtu.edu

Leo C. Ureel II
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
Email: ureel@mtu.edu

Charles Wallace
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
Email: wallace@mtu.edu

***Abstract*—We report on our efforts to enhance our undergraduate computer science and software engineering curriculum, promoting what we term *agile communication* through practice in inquiry, critique and reflection. We are targeting early courses in our curriculum, so that students internalize agile practices as part of their personal software development process. Our approach constitutes a *cognitive apprenticeship* that engages students in authentic software settings and articulates processes that are traditionally left implicit.**

Communication-intensive activities are woven through this curriculum in a variety of ways. The POGIL framework provides a structured approach to inquiry. Automated feedback on test coverage, programming style and code documentation are provided through WebTA, a novel tool that we have integrated into the Canvas learning management system, providing communication by proxy that supplements instructor feedback with continual critique of code and documentation. A program of guided inquiry through real case studies of software communication prepares students for their team software activities, and a series of reflective exercises leads them to focus on their own team communication practices.

I. Introduction: Communication in workplace and classroom

Communication — between humans — has always been a important but underappreciated aspect of software development. Although many professional software engineers are effective communicators, they typically do not have practice in articulating what it is that makes communication effective (or ineffective). That is, their knowledge remains at a tacit level, from which it is difficult to impart to students.

The situation is improving. Advocates of agile development methods in particular have always stressed the importance of flexible communication practices, deployed dynamically and strategically to maximize value. Cunningham's WikiWikiWeb, the "original wiki", contains a wealth of named patterns for agile practices, many of which fall into the domain of communication [8]. The Scrum framework is notable in this respect for the way in which it names — and therefore honors — particular communication practices that would otherwise remain tacit and invisible to students [21]. Through efforts by Sutherland, Coplein and others, Scrum rituals and principles are being captured as a pattern language of organizational techniques [19], [7].

Preparing students for the flexible, highly communicative environment of agile development is a crucial responsibility for computer science and software engineering programs in higher education. Instruction by experts in writing and communication has an essential place in the education of future software developers, but it must be matched with similar instruction within "disciplinary" courses. Computer science and software engineering instructors are in a unique position to ground the material in authentic practices, and by attending to communication in the classroom they validate it and heighten its importance in the eyes of the students.

The importance of communication in the software process is beginning to be acknowledged in the software engineering education community. The most recent version of the Software Engineering Body of Knowledge (SWEBOK) [2] has an expanded treatment of communication, with breakout sections on "reading, understanding and summarizing", "writing", "team and group communication", and "presentation skills". Recently there have been efforts to bring the expertise of writing instructors into the computer science and software engineering curricula, engage students in authentic communication activities, and categorize the genres of communication that arise in software development setting [5].

This recent addition to SWEBOK indicates that there is something else at play in software development:

> Some communication can be accomplished in writing. Software documentation is a common substitute for direct interaction. Email is another but, although it is useful, it is not always enough; also, if one sends too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise collaboration tools to share information. In addition, the use of electronic information stores, accessible to all team members, for organizational policies, standards, common engineering procedures, and project-specific information, can be most beneficial. ([2], 11-11)

The point of this passage is that software professionals must be aware of contextual factors in their design of communication. A software developer — or a student engaged in a class software project — must think both strategically

DOI 10.1109/Agile.2015.22

and tactically about the current problem at hand and the form of communication that will solve it most effectively. Having provided students the tools of their trade, in the form of authentic communication genres, instructors must give them guidelines for their wise use, based on the other individuals involved in the communication, the timing and location of the communication, and the form and style — in classical rhetorical terms, *audience*, *kairos* and *decorum* [12]). Students in computing disciplines enjoy problem solving and are well versed in principled approaches to solving technical problems. We want to give them similar tools for problem solving in the communication arena.

II. Goal: Agile communicators in software development

The principles of agile development, articulated memorably in the *Agile Manifesto* [10] as "individuals and interactions over processes and tools", "working software over comprehensive documentation", "customer collaboration over contract negotiation" and "responding to change over following a plan", have resonance throughout the software industry. With this shift comes a change in how we approach communication. Agile developers are also agile communicators, with the following strengths:

Proactivity: At the heart of the agile approach is a recognition that requirements, priorities and obstacles in software projects are in constant flux. Consequently, agile methods encourage patterns of constant questioning, informing and debating. Agile developers must be unafraid to inquire about requirements, to critique design choices, and to provide reflective comments on the team's process.

Flexibility: While agile frameworks such as Scrum and Kanban establish rituals and artifacts rooted in communication, these do not constitute a comprehensive, programmatic standard. Agile developers must be able to handle multimodal discourse (including written, oral and graphical communication through various media) and adapt to new communication situations, instead of relying on formal scripts and templates.

Creativity: In agile development, participants tailor the communication channels and genres they use dynamically to maximimize value, rather than cleave to a predefined plan. Agile developers must be skilled rhetoricians, with a deep understanding of their communication options, and an ability to choose genre and style to suit the audience and purpose.

Our goal is to build these agile communication strengths in our students, through exposure to and practice in authentic software communication settings. We wish to build this exposure and practice directly into our disciplinary courses, and early in the curriculum, so that agile communication becomes a natural part of their internalized software process. Also, by recognizing the importance of communication and engaging in it at early stages, we expect to attract and retain students who are motivated by working in teams. These students are the ideal software developers of the future, since the reality is that software development is highly social and communicative.

III. A Cognitive Apprenticeship Approach: Inquiry, Critique and Reflection

The Cognitive Apprenticeship model [6] is a constructivist approach to learning that focuses on teaching concepts and practices utilized by experts to solve problems in realistic environments. It has special relevance in the context of software development, particularly in the communicative skills that we are interested in, because it emphasizes making implicit processes explicit to the learner. In typical computer science or software engineering educational settings, topics like team communication are often deemphasized in favor of more technical topics; in the workplace, the communication-related knowledge that experienced developers possess is internalized, complicating their ability to pass it along to new employees.

We have identified three fundamental agile activities that are mediated through communication: *inquiry* (strategies for resolving unknowns, coming to a shared vision, solving problems); *critique* (systematic analysis and evaluation); *reflection* (identifying and describing one's own implicit or explicit work process). Here we explain how these three components constitute a program of cognitive apprenticeship, and how we engage our students in these activities.

A. Inquiry

Agile development demands a spirit of constant inquiry. The famous Extreme Programming admonition to "embrace change" [1] implies continuous interaction with stakeholders to understand ever-changing requirements, priorities and obstacles. The spirit of inquiry extends to intra-team communication; in a process of continual self-optimization, teams self-organize and solve problems together.

The basis for our inquiry based curriculum is the *POGIL (Process Oriented Guided Inquiry Learning)* approach, which originated in undergraduate chemistry education [9] and has been introduced to computing disciplines through the CS-POGIL initiative [15], [11]. At the heart of POGIL is a *guided inquiry* learning cycle of *exploration*, *concept invention* and *application*. Students work in small groups with well-defined roles — similarly to teams in agile software development — to encourage accountability and engagement. Each POGIL assignment has a common structure: supply students with initial data, guide them through leading questions that allow them to construct a unifying concept explaining the data, then provide means for them to apply and validate their newly constructed concept. It is in essence an application of the scientific method in a carefully crafted classroom setting. In addition to learning the core concepts at the heart of the assignment, students get practice in team problem solving and communication.

We have employed POGIL successfully in the third-year Team Software Project course, to introduce the concept of strategic communication in a software development setting [14]. 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. In POGIL, students work in small groups with individual roles — a process framework similar to that of

Category	Attribute	Possible Values/ Questions
What	Scope (Input)	What elements of the project are referenced in this communication?
	Expected outcomes (Output)	What elements are subject to change as a result of the communication? What are the expected changes?
Why	Purpose	What is the purpose of the communication? Is there an official goal to be achieved? Apart from the official goal, are there other goals or needs being met by communicating?
Who	Stakeholders	Who is involved in the communication? What are the differences between participants, in terms of knowledge, needs, status, etc.?
How	Style	Is the tone of the communication formal or informal? Is there a predefined structure in place for the communication, or is the structure to be defined during the communication itself?
	Use of artifacts	Are there artifacts (tools like written material, diagrams, code) involved? Are they physical or virtual? Who owns them? Who has access to them?
Where	Location	Is there a particular place where the communication takes place? Or does it happen virtually, in no particular place? What attributes of the location are important for the communication to be effective?
When	Duration	Is there a fixed or typical duration for the communication act?
	Synchrony	Is the communication mode synchronous (with instant response) or asynchronous (with no expectation of response time)?
	Frequency	Is there an expected or common frequency for this communication act – once a project lifetime, weekly, daily?

Fig. 1. Sample communication pattern inquiry worksheet [14].

Scrum. The problem solving conversations within the groups give students further practice in team communication. Using a simple rubric based on standard rhetorical principles of audience, purpose and style, along with structural factors such as location and timing (Fig. 1), students characterize various communication practices, then assess those characteristics with regard to particular software project settings.

As an illustration of the exploration-invention-application cycle in practice, we give an early exercise from our Team Software Project material:

Exploration. We ask students to analyze standard Scrum communication practices (*e.g.,* daily standup) that they have been exposed to earlier, and to use our rubric as a guide to identify critical features of the communication strategies used.

Invention. From these findings, students name patterns of communication and identify contextual characteristics that make the pattern suitable for application.

Application. Next, students are asked to conjecture how the nature of a communication pattern would be affected if, one by one, different attributes of the communication act were changed: *e.g.* changing the duration of the daily standup meeting to one hour, changing its frequency to monthly, or changing it to an asychronous activity with team members "checking in" remotely. Students were asked to analyze how such changes would affect other identified attributes of the communication act such as content, perceived value, and scope.

B. Critique

Agile development demands continuous attention to good design, including refactoring when changing user needs and design demands dicate. Likewise, team organization and practices are also under constant review. This requires developers to be willing and able to reassess current design and practices and to articulate areas of improvement. WebTA is a tool developed by the authors to critique student programs in introductory computer science courses. We have integrated the tool with the Canvas Learning Management System (LMS) [3] to provide immediate feedback to students.

Traditional methods for teaching computer science — lecturing on abstract concepts, assigning a programming project related to the lectures, then grading the students' submitted finished products — resemble the outdated waterfall model of software development in many ways. An instructor writes a specification and hands it off to students as an assignment. Students toil in isolation, without the benefit of instructor feedback or team communication. When they run out of time, students submit the assignment and hope for the best – not entirely sure that they interpreted the assignment in the same way as the instructor. Lastly, the instructor applies secret tests to the student work and assigns a grade, then moves on to the next topic, regardless of whether students have successfully constructed mental models to understand the current topic. Many academic programs utilize auto-graders to assign preliminary grades to student work, reducing the time burden placed on human graders. WebTA does do this, but goes further to provide constant feedback to our students in programming courses.

Through WebTA, we teach students test-driven agile development methods through small cycles of teaching, coding integrated with testing, and immediate feedback. We focus on this Learning Cycle [13] by providing students just-in-time code critiques for them to reflect on and feedback into a continous development process. This kind of style critique is based on Education Critiquing Systems [17].

Students using WebTA are engaged in communication-by-proxy with the instructor. This communication is not meant to replace instructor feedback; rather, it codifies common feedback scenarios to assist the instructor in reaching students in tight feedback loops just when the student is engaged in problem solving and learning. The instructor configures WebTA with common critiques that are triggered by errors, warnings, or textual analysis of the student's code. These critiques are issued to the students immediately, as needed by the student to support concept formation.

When students connect with WebTA through Canvas, a startup screen that explains the current problem and tells them which files they should upload to receive a code critique. After clicking on the "Critique My Code" button, students receive an online report which includes a Critique Summary. The critique summary includes a stoplight that tells the student at a glance if they succeeded in their programming task. A green light indicates a satisfactory state, an amber light indicates the presence of some minor warnings or style issues (Fig. 2), and a red light indicates serious errors.

Under the hood, the system has compiled the students' code and run it through a series of rigorous shake-down tests. Students can scroll down from the critique summary to view details of the critique, including errors and warnings generated both at compile-time and run-time. The instructor can configure the system to run both public and secret tests, run the student's own test code against their program, or assess the student's JUnit test cases to determine their ability to generate

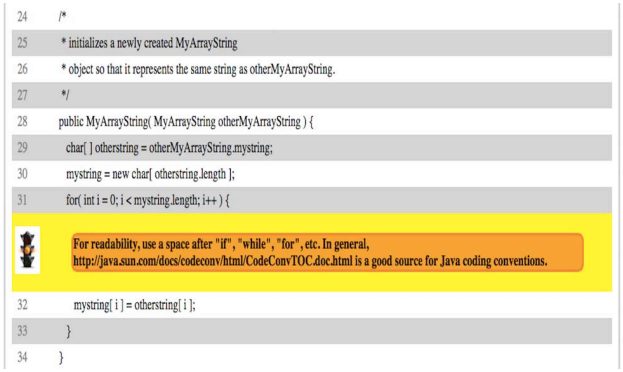

Fig. 2. WebTA style critique.

edge cases. Scrolling further down the code critique, students find a listing of each code file submitted that includes style advice generated via simple textual analysis of the code.

Students using WebTA are engaged in Learning by Doing. While developing solutions to problems, students engage in an iterative conversation: developing code, receiving critiques, reflecting on feedback, and revising their solutions. WebTA applies Cognitive Apprenticeship practices that role-model authentic skills for students. Students are repeatedly exposed to patterns of coding and critiques from which they learn how to identify and communicate about issues that crop up during software development.

C. Reflection

A key component of agile development is continual process improvement, facilitated by periodic reflective activities. The sprint retrospective in the Scrum framework, for instance, is a ritual that encourages critique and creative problem solving. The concept of professional reflection has long been touted as a valuable means of metacognitive regulation [18], and there is evidence that it builds strong teams and projects and encourages learning [16], [20].

Once our Team Software Project students have been immersed in the Scrum cycle, iterating through multiple sprints, we ask them to reflect on their own process from a communication perspective. In the "How We Scrum" activity, they identify commonly recurring communicative acts as patterns and assess their effectiveness. Through this activity, they come to acknowledge that they have created a communication infrastructure of their own design. They then critique this design and propose improvements.

Later, in the "How They Scrum" activity, student team members interview members of other project teams about their communication practices. This contrasting perspective emphasizes the fact that they, and the other teams, have made choices that affect project performance. Sample reflective comments like those shown in Fig. 3 show the influence of earlier POGIL investigations of communication strategies; in their reflections, students use the pattern approach to discuss their own communication design choices [14].

> Our communication style and format is largely based on face to face meetings several times a week whereas the other team depends much more on google docs to keep each other up to date. This face to face format keeps the frequency of our communications to a set number and time/place whereas the other team is in asynchronous communication constantly. For example, we find it beneficial to meet and discuss how to approach a problem, divide up the work, and then part ways to work on it separately, coming together again for our next meeting to discuss our progress. We find this helps keep each other accountable for the work that needs to be done. In contrast the other team does this much less, but updates google docs much more frequently. This allows them to spend whatever time they have available on a given task and work more independently

Fig. 3. Sample student reflections: "How They Scrum" [14].

D. Cognitive apprenticeship

Collins [6] outlines the elements of a general framework for cognitive apprenticeship environments:

Content: Two types of content need to be taught to students: Domain Knowledge and Strategic Knowledge. *Domain knowledge* consists of the technical topics normally taught in computer science classrooms: *e.g.,* programming languages, data structures, algorithms. *Strategic Knowledge* is what experts use to make use of these classroom skills to solve real-world problems. Strategic knowledge is often difficult to express in the classroom because it is founded in experience gained from doing computer science. Here we outline the synergistic relations between Collins's areas of strategic knowledge and our approach.

Heuristic Strategies: We support student learning through the use of patterns in communication and learning [14], [4]. Software professionals routinely use sophisticated problem solving and design skills in their communication with one another and other stakeholders in the software process. We wish to impress upon the students the importance of communication in software development, and to encourage strategic and tactical thinking about communication.

Control (Metacognitive) Strategies: Through reflection stimulated by WebTA critiques and POGIL exercises, we encourage students to learn from choices they, their teammates, and other teams make during the development process.

Learning Strategies: In our classrooms, we engage in role-modeling, role-play, and POGIL activities to help students *learn how to learn*. The cycle of doing, critiquing, reflecting, and redoing helps students develop their own learning strategies and apply them to problem solving.

Method: Students using WebTA are *scaffolded* in the continuous development cycle with automated testing. Students are *coached* through code critiques and helpful suggestions that prompt the to *reflect* and refactor.

Our POGIL approach, with its emphasis on communication falls squarely within Cognitive Apprenticeship methods. We invite students to be *articulate* communicators, to *reflect* on their communication choices, and to *explore* new ways of approaching problems in a team setting.

Sequencing: *Increasing complexity* refers to the presentation of topics and the learning of skills in a way that builds increasingly towards expert performance. *Increasing diversity* is the

sequencing of learning tasks such that a wider range of skills are increasingly required to solve problems. *Global before local* involves introducing students to high level concepts and working towards detailed implementations.

We introduce POGIL strategies in our initial computer science courses. By teaching students Test-Driven Development from day one, we set students on a course to being agile developers. We then introduce students to *user stories* and the notion that communication is a critical component of software development. When students reach the Team Software Project course in their third year, they are ready to learn POGIL strategies for communication and problem solving.

Sociology. Traditional teaching methods in computer science produce graduates with classroom skills and knowledge but no context or experience for applying those skills; sadly, in many cases real learning only starts when students leave academia and are faced with real-world problems. It is critical that cognitive apprentices be involved in solving real-world problems using real-world techniques as soon as possible. The Team Software Project course provides *situated learning* by having students work on authentic problems and communicate with stakeholders in the same way they will on the job. Here they engage in a *Community of Practice* [22] where they actively communicate and use the skills introduced in earlier courses. The POGIL methodology *exploits cooperation* extending learning and providing *intrinsic motivation*.

IV. Conclusion

Our inquiry and reflection exercises in the Team Software Project course have been in use for the past three years, while the WebTA code critique tool has only been in place for a year in our introductory programming courses and our Data Structures course. We have reported earlier on the effectiveness of the Team Software material in heightening students' awareness of strategic communication and giving them tools for reflection and process improvement [14]. We give a brief summary of some preliminary conclusions concerning our project.

Early experience with WebTA's proxy critique functionality has revealed some unintended consequences that may require fine tuning. Introductory Programming students seem to have learned more debugging skills in Fall 2013, before WebTA was in place. In Fall 2014, students relied on WebTA's shakedown testing to provide information for debugging and as a result acquired fewer debugging skills. Data Structures students were required to submit JUnit test cases with their code during both Fall 2013 and 2014 semesters. WebTA tested their JUnit tests against the assignment API. Over the course of Fall 2014 we saw marked improvement in student conformance to the specified API. However, we also noticed a tendency for students to drop tests if they did not understand an edge case. At this preliminary stage, we have the sense that more effort needs to focus on fading scaffolds and teaching students how to test their code.

In survey results at the end of the Team Software Project course, students said they would have liked to see even more communication based activities. Some students said they would have liked to experience simulations of industry type communication that they would not normally get to experience, like attending a conference call, so they would be better prepared when they experience it in industry. They would have also liked to participate in more role playing exercises where they have to think creatively and make communication choices, like understanding how to report unpleasant news to their boss or dealing with an unproductive team member. These results give us confidence that there is a place for explicit discussion of communication in our disciplinary courses.

References

[1] K. Beck and C. Andres (2004). *Extreme Programming Explained: Embrace Change* (2nd edition). Addison Wesley.

[2] P. Bourque and R.E. Fairley (eds.) (2014). *SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society.

[3] Canvas Learning Management System. `canvaslms.com`

[4] J.M. Carroll and U. Farooq (2007). Patterns as a paradigm for theory in community-based learning. *International Journal of Computer-Supported Collaborative Learning* 2(1), 41-59.

[5] M. Carter, G. Gannod, J. Burge, P. Anderson, M. Vouk and M. Hoffmann (2011). Communication genres: Integrating communication into the software engineering curriculum. Proceedings of Conference on Software Engineering Education and Training 2011, Honolulu HI, 21–30.

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

[7] J.O. Coplein and N.B. Harrison (2004). *Organizational Patterns of Agile Software Development*. Prentice Hall.

[8] W. Cunningham. Wikiwikiweb: People Projects and Patterns. `c2.com/cgi/wiki?PeopleProjectsAndPatterns`

[9] T. Eberlein *et al.* (2008) Pedagogies of engagement in science: a comparison of PBL, POGIL, and PLTL. *Biochemistry and Molecular Biology Education* 36: 262–273.

[10] M. Fowler and J. Highsmith (2001). The agile manifesto. http://www.canvaslms.com/http://www.canvaslms.com/*Software Development* 9(8): 28-35. `agilemanifesto.org`

[11] H.H. Hu and T.D. Shepherd (2013). Using POGIL to help students learn to program. *ACM Transactions on Computing Education* 13(3).

[12] G.A. Kennedy (1991). *Aristotle, on Rhetoric. A Theory of Civic Discourse*. Oxford University Press.

[13] A.Y. Kolb and D.A. Kolb (2005). *The Kolb learning style inventory–version 3.1 2005 technical specifications*. Boston, MA: Hay Resource Direct, vol 200, 2005.

[14] S. Kumar and C. Wallace (2014). Instruction in software project communication through guided inquiry and reflection. Proceedings of Frontiers in Education 2014, Madrid, Spain, 1–9.

[15] C. Kussmaul (2012). Process oriented guided inquiry learning (POGIL) for computer science. Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh NC, 373–378.

[16] M. Lamoreux (2005). Improving agile team learning by improving team reflections. Proceedings of AGILE 2005, Denver CO, 139–144.

[17] L. Qiu and C. Riesbeck (2008). An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer. J. of Interactive Learning Research, 2008, 908–916.

[18] Donald Schön (1990). *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions*. Jossey Bass.

[19] Scrum Pattern Community. `www.scrumplop.org`

[20] D. Talby, O. Hazzan, Y. Dubinsky and A. Keren (2006). Reflections on Reflection in Agile Software Development. Proceedings of AGILE 2006, Minneapolis MN, 11 pp.–112.

[21] C. Wallace, S. Mohan, D. Troy and M. E. Hoffman (2012). Scrum across the CS/SE curricula: A retrospective. Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, Raleigh NC, 5–6.

[22] E. Wenger (1998). *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press.