Active learning in introductory programming courses through Student-led “live coding” and test-driven pair programming

Alessio Gaspar, Sarah Langevin
University of South Florida, 3334 Winter Lake Road, 33803 Lakeland, FL, USA
[alessio | sarah] @ softice.lakeland.usf.edu

Abstract

This paper revisits two emerging active learning practices in introductory programming courses and proposes ways they can be further improved. We first focus on a category of assignments which can further support the switch from instructor-led to student-led live coding practices, thus helping develop the former into a real active learning pedagogy. Then, we propose to leverage test-driven development techniques through assignments meant to engage students in competitive learning without the drawbacks usually associated with this type of programming competition framework. The new activities have been tested in courses taught at the University of South Florida and observations of their impact are discussed with respect to constructive alignment theory, constructivist educational approaches, discovery learning and pair programming / test-driven techniques.

Keywords: Computing Education Research, CS-1, Introductory Programming Courses, live coding, active learning, pair programming, test-driven development

1. INTRODUCTION

Problem Statement

The pedagogy of introductory programming courses is one of the most prolific, if not controversial, topics in the relatively new computing education research community. The sheer number of publications devoted to this single topic, as well as the traffic generated on the ACM’s SIGCSE mailing list by related discussions, both confirms the importance of CS-1 pedagogical research in our community.

Among the plethora of teaching and learning approaches which have been formulated, active learning techniques caught our attention. The very idea of active learning is to engage students in various activities which aim at facilitating the acquisition of new knowledge and skills in a learner-centered manner. These practices are inspired by constructivist theories in educational psychology and typically involve activities such as case study, reflections on class notes, group discussions, hands-on experimentations, etc. These also each promote, to a certain extent, a discovery learning approach in which the students, under the guidance of their instructors, (re-)discover autonomously the knowledge units they are being taught. It is not the objective of this paper to delve in the theoretical foundations of such approaches. Instead, we will address the limitations of commonly used active learning approaches to teaching programming and discuss how to improve them.

Objectives

The specific issue this paper is concerned with is the improvement of newly introduced active learning practices in introductory programming courses. To this end, we will review, and improve on two categories of classroom activities: “live coding” and test-driven development.

While a significant amount of work has already been devoted to these topics, we found the two above-mentioned practices to be poorly documented in the literature. We share our experience in developing new aspects of these activities and discuss their implementations in two introductory programming courses taught at the University of South Florida (USF). These two courses are interesting to lead such a preliminary study in so far that they gathered a diverse student population (non-traditional age groups, diversified majors, various levels of preliminary exposure to programming…) and employed a variety of languages ranging from flowcharts interpreters **[1,2,3]** to Java **[4]** and C **[5]**. Our discussion provides insights as to the best practices in implementing similar activities, singles out and details a specific assignment in both categories, and review results at the light of underlying educational theories

Paper Organization

Section #2 will discuss how live coding activities, which are traditionally mostly instructor-led, are being converted into genuine active learning practices by the computing education community. We will go further in this direction by examining how certain categories of assignments help improving the pedagogical impact of such practice. Section #3 will then propose to leverage test-driven development techniques and pair programming practices in the classroom to engage students in a form of competitive learning focused on code quality as opposed to encourage quick, dirty but efficient coding. Section #4 will then conclude by summarizing this paper and discussing our related ongoing work.

2. STUDENTS-LED LIVE CODING

The term “live coding” is often used to refer to a teaching practice by which the instructor exposes the programming thought process leading him to solve a given problem by programming “live” while connected to a data projector. This section focuses on improving a more recent variant which we refer to as student-led live coding

Motivation & innovative aspects

The qualitative leap, from simply showing complete solutions to students on a slide to detailing the process which leads to them by coding it “live” in front of the class, significantly improves the pedagogy of teaching programming. However, this practice is still mostly passive form the students’ point of view. The fact that some of these live coding sessions have been recorded by instructors so that students can view them out of class is further indication of the inherent lack of interactivity in this kind of activity. Therefore, while the objective of exposing the programming thought process is reached, we are still fundamentally dealing here with a passive learning strategy.

Given the advantages of active learning and of live coding, it is tempting to devise a way to combine their respective benefits into a single activity. This is essentially the idea which was originally mentioned on the SIGCSE-members mailing list almost a year ago **[6]**. The innovative aspect is to switch the focus of the activity away from the instructor and assign to a student the role of leading the live coding of a particular assignment. How is this different from simply assigning students exercises which are incorporated to the lecture material **[7, 8, 9]**? For instance, the ubiquitous presenter team at UCSD **[10]** already explored the use of specific instructional technologies (tablet PCs, web browsers, UP software) to enable instructors to collect digital submissions during classes, pick some, and show them back to the entire class in order to comment, correct or simply annotate them.

The difference is essentially similar to the one existing between instructor-led live coding and simply showing slides with the complete solution to an assignment to students. In both cases, we are not only interested in the end product students come up with but rather in the way they produce it. Just as instructor-led live coding exposes the instructor's programming thought process for students to learn from, the student-led version expose the students' thought process for the instructor to provide feedback on. The parallel goes further. During instructor-led "live coding", the errors and hesitations are important in so far that they show students that;

(1) The programming process is not linear, nobody writes 5000 lines of code without jumping back and forth in the code to adjust things and…

(2) Making errors happens to all of us but catching them in a timely manner is what will ultimately impact the quality of the resulting code.

The same goes with students-led live coding with one major difference; the nature of the errors an instructor is likely to commit is different from the nature of the errors and difficulties students might come across. Our hypothesis is that because students will relate more closely to their classmates' difficulties they will also learn more from seeing them exposed and criticized by other classmates or the instructor. We also believe this approach to be more natural than having the instructor fake errors during live coding. Students-led live coding is therefore complementary and addresses different learners' needs.

Going a little further

Our focus has been so far on the live coding activity from the students' perspective. It might be useful to change focus in order to further improve this practice. How can we improve the students-led live coding without modifying the process itself?

The previous sections didn't detail the nature of the problems the students would have to solve in such live coding sessions. Most readers probably assumed that most textbook exercises would fit the bill. However, we would like to differentiate between two categories of programming exercises; those who have a single correct solution and those who feature multiple functional solutions differing at the design or programming style levels. Considering the former, very little can be done during these live coding sessions besides identifying the syntactical and logic errors committed by the student performing the live coding. While this is already extremely useful from a pedagogical standpoint, let's consider the benefits of assigning the second category of problems to the students

Implementation & Best practices

This approach was implemented in two courses taught at the University of South Florida (USF). In order to put our discussion in the appropriate context, we will start with a brief review of their defining characteristics.

(1) "Programming Concepts" (cop2510) is meant for both students who already took an introductory programming course at a community college as well as for complete neophytes. This course is usually taught in the department with an object-first approach and relies on either the Java or Python programming languages. When teaching the course, this author has been using Java and the BlueJ development environment **[4]**.

(2) "Program Design" (cop3515) exists in two versions; computer science and information technology. In both versions, this course requires students to pass with a minimal grade prior to being accepted in either program. Typically, students taking this course had already several programming courses at community college level including cop2510. This semester, the course was taught using the C language in a Linux environment. The objective was to articulate the transition between introductory programming courses and system-oriented upper level courses such as operating systems.

In this context, student-led live coding has been implemented with different objectives in mind. For cop3515, the technique has been used to first introduce recursion to our students. The other topics covered in the course were technical in nature (stack, heap, memory management) but offered difficulties of a different nature. The assignment presented in the following question is the one we used in this course. Concerning cop2510, the technique has been used in a more classic way, letting students work on design-from-scratch problems involving the definition and implementation of several classes.

What are the lessons learned so far? The first lesson learned is one of conviviality and classroom dynamics. Our first attempt at focusing the live coding activity on students resulted in picking a student, having her walk to the podium PC connected to the data projector and let her work in front of everyone. For many students, this contributed to reduce the participation to a strict minimum as they felt "singled out" and uncomfortable standing and working in front of the entire class. This issue was addressed this semester by ordering a wireless keyboard-touchpad device which could be easily passed among seated students as suggested in the original SIGCSE-members mailing list posts. A 2.4 GHz RF device allowed an increased range of operation thus annihilating the secure feeling of the students populating the back rows. It also helped considerably to have students connect to their own accounts on a server instead of having to work on the podium PC and then get their results on a thumb drive. This was no problems with cop3515 since students' accounts were hosted on a Linux server. Each student selected to work on the next live coding exercise, simply logged in using an SSH client and a X-server (X-Ming) and started working in her own environment. In cop2510, as we used BlueJ, each student had to work on the podium PC locally. This issue can however be addressed by using the plug-in developed for BlueJ which allow the IDE to seamlessly access files stored in a student account on a remote Linux server **[11]**.

Another interesting practical question is the choice of the next student to participate. During the first couple of sessions, the choice is random. However, as your knowledge of their strengths and weaknesses grows, you will be faced with the dilemma of either selecting students which will serve as "model" to others or students who are in difficulty with the current material. From our personal experience, the former category is most beneficial during the first couple of live coding sessions. It is difficult to convince students of the usefulness of such an activity if the first attempts turn into an embarrassing silence. On the contrary, a motivated student, regardless of the quality of her coding, will actually help introduce the activity and reduce the stress levels of those who will soon be participating in it. After one or two sessions, the choice of who goes next is almost irrelevant in so far that the routine will be established, and students will most likely already realize what they can get from these exercises. At this point, each difficulty encountered by the live coding student, each alternative solution they will come up with, becomes a hook for the instructor's to informally introduce the lecture material in a much more problem-solving based and thus less boring manner.

Assignment example

The following assignment has been given in the cop3515 "program design" class using the C language. It took place within the first 4 weeks of the course and was part of a series of questions focused on recursive programming. Here is how the assignment read;

Implement an iterative and recursive version of a function which will return how many times its (strictly positive integer) argument can be divided by two until you get a non null remainder. Examples;

F (4)	→	2 time(s)
F (5)	→	0 time(s)
F (6)	→	1 time(s)

This assignment is purposely open to interpretation but after a small lecture on recursion, a handful of classic examples and a couple of take home exercises, our students were ready to innovate (or realize they had to spend some more time on the topic). Here are the various results we obtained for this exercise;

Classic iterative solution:

```
int F (int no)
{
        int count = 0;
        while (no % 2 == 0)
        {
                no /= 2;
                count++;
        }
        return count;
}
```

Classic recursive solution, the result is built as the recursive calls return (this is the solution which was expected from students due to its similarity with the lecture examples). It is interesting to note that while developing this solution, some student suggested making the recursive call as: *F (no/=2);* while not strictly incorrect, this kind of remark is a good starter for a class discussion and explanation of what superfluous code can mean. The stack diagram was used to show the student how this difference impacts the local variables. This helped in realizing that although not leading to a bug per se, this approach would be comparable to walk in a direction taking 3 steps forward and 1 step back.

```
int  F (int no)
{
        if (no % 2 == 0)     return 1 + F (no / 2);
        else                 return 0;
}
```

The following solution illustrates how the result can be constructed while making the recursive calls instead of as they are returning. This solution wasn't originally planned for this lecture and a student came across it. Explaining this new possibility in details (using stack diagrams) helped reinforce the understanding of the classic recursive solution. This was true even with students who didn't think of this alternative at first or those who needed a refresher on the working of the stack.

```
int   F (int no , int count)
{
        if (no % 2 == 0)     return F (no /= 2 , ++count);
        else                 return count ;
}
```

Finally, the two last solutions really surprised us. The lecture on recursion took place right after explaining variables scope and duration based on their location in the stack and heap segments. Some students, armed with this freshly acquired knowledge, immediately saw a way to apply it to the problem at hand. While discussing incorrect versions of the above functions, one bug caused the count to never be modified either when passed to the recursive call or when incremented before to be returned. This motivated some students to come up with a fix which led to the following solutions using respectively a global variable or a static local variable;

```
int counter = 0;
int   F (int no)
{
        if (no % 2 == 0)
        {
                counter++;
                return F (no / 2);
        } else
                return counter;
}
```

```
int    F (int no)
{
        static int counter = 0;
        if (no % 2 == 0)
        {
                counter++;
                return F (no /= 2);
        } else
                return counter;
}
```

These solutions allowed for a discussion of the potential problems that could emerge from exposing a global variable when other programmers would try to reuse this code.

Discussion

The above example showed that using assignments with multiple correct solutions in the context of student-led live coding has an interesting pedagogical potential and enhances further this type of practice. Several observations resulted from our first experimentations with this approach. At first, we believe a carefully crafted series of assignments would allow for the entire lecture contents to "emerge" in a didactic manner rather than being explicitly stated through a slide show. It was really rewarding to witness students'' remarks prompt for the answers that constitute the material of a traditional lecture on the topic. While this is the fundamental characteristics of any active learning methodology, we also believe that these particular assignments actually bordered on discovery learning **[16]**. It is our intent to explore further the theories and practical applications of this approach to improve our future work on student-led live coding.

Secondly, this activity also enabled students to not only apply or adapt, through analogical thinking, the material introduced in the preceding lecture but also to generalize it. Some beginning programmers are on the outlook for a “book of answers” which would associate to every possible exercise or assignment, a solution template they could memorize and regurgitate at the exam in order to get a passing grade. This attitude and misconception of what programming is, often finds its root in either a fundamental inability to grasp the nature of the programming activity or, sometimes, in the bad habits learned in a first-programming course which requested students to modify existing code through cut and paste operations all semester long. The above-mentioned assignment was given in class after a short lecture on recursion which illustrated the principle with the help of the classical factorial and Fibonachi numbers examples only. It is interesting, given this limited set of examples, to see how the class, as a whole, ended up going well beyond the direct application of these patterns to solving new problems. Once the initial attempts at defining F (n) in terms of F (n-1) instead of F (n/2) failed, students started generating a diverse spectrum of solutions to this simple problem. This is the clear sign of ongoing cognitive processes which go beyond straightforward analogy-based thinking but attempt to generalize the knowledge provided in the lecture into new ways to develop recursive solutions. Quite naturally, some of these solutions were less desirable than others but they all contributed nonetheless to instruct the entire class on both what is and what is not appropriate. In more formal terms, our students have been moving along the SOLO (Structure of the Observed Learning Outcome) taxonomy **[17,21]**.

We also observed that this type of activity helps students gain a better understanding of the grading process. Previous work already stressed out that many students have misguided conceptions about the very idea of program correctness **[12]**. For some, a successful compilation means that the job is done and the program is ready to be turned in. Only few beginning programmers will actually test their code at runtime in various scenario and even less will understand at first that the test harness must be carefully crafted to tell us anything at all about the program’s correctness. Addressing such misconceptions is clearly an important learning outcome for a programming course and one that can’t be tackled too early. When engaging students in instructor-led live coding, a good example of the expected product and process is already provided. This helps them understand what they will be expected to produce. However, with student-led live coding, many more opportunities for the instructor to correct students’ code “on the fly” emerge. Quantitatively, the sheer amount of opportunity for the instructor to provide feedback to students makes this approach worthwhile. Not only can the errors made by the live coding student be addressed and discussed in class, but the her classmates’ suggestions can also be used to detect and address misunderstandings. The lecture then almost emerges from these interactions and provides both students and instructors with a much more motivational framework as compared to lecture-only scenarii. From a more qualitative standpoint, each correction from the instructor provides an opportunity for students to get a glimpse at the way program correctness is evaluated by a more experienced programmer. Not only that, but this same evaluation process is most likely the one that will be employed to grade their assignments and exams. Therefore, student-led live coding is not only about the instructor gaining a better understanding of his students’ cognitive processes but also about the students themselves gaining a better understanding of how the instructor perceives and validates their work. This goes a long way toward reconciling the courses objectives, expectations, examination modalities and teaching practices. According to the proponents of the constructive alignment theory **[13]** students with diverse motivations can be effectively channeled into learning the very set of skills the course is targeting by ensuring an appropriate overlapping of this skill set and the skills necessary to actually simply pass the exam. In our context, we demonstrate continuously the concrete expectation for both the final product (code) and its development process thus allowing students to adapt to match these over the course of the semester.

Finally, it has been already said that during student-led live coding sessions, the rest of the class tends to assume the role of the observer as defined in pair programming practices **[14,15]**. Pair programming has already been shown to have an interesting impact on programming courses’ pedagogy. It is therefore very positive for student-led live coding to trigger some of the dynamics observed in these studies. What we observed during our practice is that the direct neighbors of the live coding student seem to generally assume a higher degree of responsibility regarding the quality of the produced code. While this is not a systematic occurrence, it seems that once a student is selected as the live coder for the next assignment, his neighbors feel that this is their responsibility to convey feedback to their classmate. The physical proximity means that their interventions can be a simple whisper, nod or finger pointed at the screen as opposed to the way other classmates will have to “stand up” to participate. We believe that this is a result of an all too common tradition of holding back during in-class participation to avoid embarrassment. However, because of this effect, the instructor can, if the entire class isn’t participating enough, solicit feedback from the students most likely to volunteer it; the live coder’s neighbors. This also suggests that, if possible at all, re-arranging the sitting arrangement of a computer lab might lead to better participations. While this is not a novel idea in itself, very few classrooms offer this type of flexibility to instructors in the computing disciplines curricula.

3. ANTAGONISTIC LEARNING ACTIVITIES

In the previous section, we mentioned that some behaviors observed during student-led live coding sessions relate to the “observer role” defined in pair programming terminology. This section further explores how pair programming can be enhanced through test-driven development practices in order to create early assignments, meant for introductory programming courses, which engage students in a quality-focused form of competitive learning.

Motivation & innovative aspects

The literature on pair programming in a CS1 course is abundant **[14, 15]** and already identified many pedagogical benefits of this approach. Pair programming can mainly be seen as a collaborative learning strategy. In contrast with such approaches, mini-games such as Robocode **[22]** have been designed to facilitate the learning of programming by engaging students in competitions during which their programs will be evaluated against one another in an arena of sorts. These two different, and somewhat antagonistic, approaches to raise students’ motivation level inspired us to revisit the pair programming fundamental idea and twist it to introduce a competitive dynamics. Our early experience with Robocode indicated that such approaches have the potential to motivate students to focus too much on the final result (i.e. a tank which can "blast its way" to victory) rather than the programming process. In a typical AI course, this effect would manifest through the fact that the code winning the competition would be a better illustration of how human can "play the system" rather than a solid implementation of one of the artificial intelligence approaches discussed in the lecture. In the context of a first programming course, it would be most likely the code quality which would suffer from the imperative necessity to win. Interestingly enough, Ken Schwaber stressed out that sacrificing code quality is almost a "second nature", an instinct for developers caught in a goal-driven, pressured productivity environment **[20]**. Obviously, we might want to steer away from nurturing such reflex in beginning programmers.

Going a little further

How can we introduce a competitive learning drive along with pair programming while avoiding such an educational pitfall? Our suggestion is to switch the focus from the end results (e.g. "my code is ready before yours") to the process and more specifically to code correctness. This allows the instructor to introduce the notion of test harness and have students start, from the very beginning of the course, to appropriately test the code they produce. As mentioned in **[12]** this is not a goal which importance should be underestimated.

We tested, in the previously mentioned courses, a variant of the pair programming activity based on test-driven practices. We asked two students to start by both developing independently a solution to the same assignment. Once their solutions were coded, we introduced the idea of test harness to them by asking them to verify how their code functioned with a diversified set of inputs. After this second step, they were required to team up with their neighbor, exchange seats and start working on testing their classmate's code. Our goal is to lead students to start thinking *during the development phase* in terms of "how would my code react to this input?" and "what is a programmer most likely to overlook in this method?". The evolution of CS1 programming courses contents indicates that programming is now perceived as an activity that goes beyond the mere writing of code but requires many skills. Designing, implementing, testing and debugging are all examples of the programmers' skill set. The activity we propose in this section helps sharpening multiple skills of this set without focusing only on the purely implementation-focused ones.

This switch of the competitive focus away from the resulting code's efficiency and more toward its correctness is extremely valuable, from a pedagogical standpoint, for early programming courses. It is also similar to the way instructor or student-led live coding contributed to aligning what is taught and the way it is taught with the expected outcomes of a programming course: teaching students how to program.

Implementation and best practices

Our objective was to leverage test-driven development to introduce beginning programming students as early as possible to the concept of test harness and engage them in a competitive activity to further motivate the development of a different attitude toward coding which we might define as "defensive coding". This attitude is meant to enable a developer to code while thinking in terms of code testing. We didn't aim at making students think in terms of test-driven development per se but rather at re-enforcing good programming practices from the get go. For this reason, we used the Raptor flowchart interpreter **[1, 2, 3]** in both above-mentioned courses. In cop2510 we used it for about 3 weeks prior to a BlueJ-based object first approach. In cop3515, we used it for only 1 week and a half as a refresher and review material. Raptor allowed us to spare our students the learning of a specific syntax during the first weeks but instead focus on developing solid control flow design skills. In this specific context, we developed the assignment presented below.

Assignment example

This exercise has been adapted from one of Nick Parlante's Javabat applets available at http://javabat.com/:

The squirrels in Palo Alto spend most of the day playing. In particular, they play if the temperature is between 60 and 90 (inclusive). Unless it is summer, then the upper limit is 100 instead of 90.

Using raptor, write a flowchart which is going to ask the user to provide a temperature value (between 0 and 130) and a number summer which will be equal to 0 or 1. Depending on the values that were passed to you by the user, you will determine whether the squirrels are playing or not and display on the screen an appropriate string.

This first step was followed by an individual code testing;

Make sure you test extensively your program. This time you will write down the tests you have been performing on paper as follows:

A table was provided for students to design and then record their testing experiments.

Value for TEMP	Value for SUMMER	Expected outcome	Observed outcome

Finally, the next exercise extended the testing to the neighbor's code. At this point the objective was clearly to attack each other's code through the test harness in order to "prove" it wrong and make your own code look better. Good corporate practice altogether, isn't it?

Now that you developed both a flowchart and a series of test cases to make sure your program works, exchange seats with your neighbor and run your tests on their program. Try to find tests which will prove their code wrong, keep track of these results and show them to your classmate once you think you can't find more bugs in their code for them to fix it.

Discussion

Regardless of the students' preliminary programming experience, this activity allowed them to play both the developer and observer roles on each assignment rather than taking turn. This might be beneficial when trying to introduce testing at such an early stage in so far that students don't have to understand and adapt to a role description. More importantly, this assignment introduced the testing mindset as a competitive game which is more likely to motivate students and which we hope will help them develop a solid programming thought process which will address the issues stressed out in **[12]**.

4. Discussion & Future work

This paper discussed two active learning techniques appropriate for introductory programming courses. The first one enhances the practice of student-led live doing by employing a particular type of assignment. We discussed the benefits of this approach in terms of its connection to the constructive alignment theory and, more generally speaking, in terms of constructivism and active learning. The second revisits the pair programming pedagogy and leverages competitive learning in a code quality focused context by using test-driven development methodologies.

Our future work will be concerned with evaluating quantitatively the impact of the above-mentioned approaches. As of the writing of this paper, we are still in the process of collecting anonymous feedback from our students and hope to be able to include early evaluations in the final manuscript. After this first evaluation, we will refine our surveying tool in order to focus more carefully on the strengths and weaknesses of these approaches as identified by students.

Our next step will then be to work on scaling up these results; like many active learning activities, ours were successful in a small size classroom context but will need further work if they are to scale up to larger groups of students. The activities themselves and the very idea of student-led live coding might have to be revisited from the perspective of groups of students working together instead.

5. Acknowledgement

The authors would like to thank the cop3515 students for their participation in the students-led live coding activities and the original designs they produced during these sessions which have been used to illustrate this type of activity in this paper.

6. References

[1] M.C. Carlisle, T.A. Wilson, J. W. Humphries, S.N. Hadfield, “Raptor: a visual programming environment for teaching algorithmic problem solving”, Proceedings of the 36th SIGCSE technical symposium on Computer science education, St Louis, Missouri, USA, 2005, pp. 176-180

[2] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, Steven M. Hadfield, “Raptor: introducing programming to non-majors with flowcharts”, April 2004, Journal of Computing Sciences in Colleges, Vol.19 Issue 4

[3] J. C. Giordano, M. Carlisle, “Tools and systems: Toward a more effective visualization tool to teach novice programmers, October 2006, Proceedings of the 7th conference on Information technology education SIGITE

[4] M. Kolling, B. Quig, A. Patterson, J. Rosenberg, “The BlueJ system and its pedagogy”, Journal of Computer Science Education, special issue on learning and teaching object technology, vol 13, no 4, 12/2003

[5] A. Gaspar, A. Ejnioui, N. Boyer, “The role of the C language in modern computing curricula", in preparation

[6] M. Hailperin, SIGCSE-members mailing list, posts #34 and #37, July 10th 2006, accessed on 2/22.2007 at http://listserv.acm.org/archives/sigcse-members.html

[7] R.M. Felder, “It goes without saying”, Chem. Engr. Education, 25(3), 132-133 (Summer 1991), accessed 2/22/2007 at http://www.ncsu.edu/felder-public/Papers/Education_Papers.html

[8] M. Kolling, D.J. Barnes, “Enhancing apprentice-based learning of Java”, 35th SIGCSE technical symposium on computer science education, 2004, pp. 286-290

[9] O. Astrachan, D. Reed, “AAA and CS 1: The Applied Apprenticeship Approach to CS 1”, Proceedings of SIGCSE, 1995

[10] R. Anderson, R.Anderson, O. Chung, K. M. Davis, P. Davis, C. Prince, V. Razmov and B. Simon, “Classroom Presenter – A classroom interaction system for active and collaborative learning, workshop on the impact of pen technologies on education”, 2006

[11] Bluej SSH access plugin, http://www.bluej.org/

[12] Yifat Ben-David Kolikant, “Students' alternative standards for correctness”, Proceedings of the 2005 international workshop on Computing education research ICER '05

[13] J. Biggs, “Teaching for Quality Learning at University”, Buckingham: Open University Press/McGraw Hill Educational, 1999, 2003.

[14] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, S. Balik, “Improving the CS1 experience with pair programming”, January 2003, Proceedings of the 34th SIGCSE technical symposium on Computer science education SIGCSE '03, Volume 35 Issue 1

[15] C. McDowell, B. Hanks, L. Werner, Experimenting with pair programming in the classroom, June 2003, ACM SIGCSE Bulletin , Proceedings of the 8th annual conference on Innovation and technology in computer science education ITiCSE '03, Volume 35 Issue 3

[16] J.S. Bruner, “The Process of Education”, Cambridge, MA, Harvard university press, 1960

[17] R. Lister, B. Simon, E. Thompson, J. Whalley, C. Prasad, “Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy”, Innovation and Technology in Computer Science Education (ITiCSE), 2006.

[18] J. Bennedsen, M.E. Caspersen, Revealing the programming process, Proceedings of the 36th SIGCSE technical symposium on Computer science education, St Louis, Missouri, USA, 2005, pp. 186-190

[19] E. Frank Barry, Christopher C. Ellsworth, Barry L. Kurtz and James T. Wilkes, “Teaching OO Methodology in a project-driven CS-2 course”, Conference on Object Oriented Programming Systems Languages and Applications, 2005, pp. 338-343

[20] K. Schwaber, “Scrum et al”, Google tech 9/5/2006, accessed 2/22/2007 at http://video.google.com/videoplay?docid=-7230144396191025011&q=google+tech+talks

[21] J. Biggs, K. Collis, “Evaluating the Quality of Learning: The SOLO Taxonomy”, New York : Academic Press, 1982.

[22] Kevin Bierre, Phil Ventura, Andrew Phelps, Christopher Egert, "Motivating OOP by blowing things up: an exercise in cooperation and competition in an introductory java programming course”, March 2006, Proceedings of the 37th SIGCSE technical symposium on Computer science education