An Experience Report on Improving Constructive Alignment in an Introduction to Programming

Alessio Gaspar & Sarah Langevin
University of South Florida Polytechnic, 3433 Winter Lake Road, FL 33803
Contact Author: alessio@poly.usf.edu

ABSTRACT

This paper discusses the need for learning activities adhering to the constructive alignment theory in introductions to programming. Problems with widespread instruction methods are used to motivate a new variant of pair programming which reduces the possibilities for students to "evade" practicing the very skills pair programming is expected to allow them to hone. We present an example of an exercise integrating the new pedagogy and discuss its initial reception by students.

1. Introduction

In educational research studies, interventions are often distinguished by whether they affect pedagogy of content or pedagogy of instruction. The former focuses on what topics are being taught, in what order, how they link to one another; e.g. objects-first vs. fundamentals first. The latter is more concerned with how these topics are taught; e.g. active learning vs. traditional lecturing. While most research is devoted to assessing the impact of new instruction methods, the pedagogy of content seems to be seen as not warranting further research beyond the object / class first debate. However, regardless of the chosen path, many textbooks share the same pedagogy when it comes time to impart to students the skills to design and implement programs from scratch based on requirements. Among these common traits, two have been the focus of our attention in this study;

- **P1 –** The programming thought process – referred to as PTP henceforth – is presented as being sequential. First, a design is defined then, it is implemented and tested. During the design phase, a list of high-level steps is identified. Each step is then decomposed into smaller steps which are further decomposed into even smaller steps. This top-down refinement process is applied until the original list of steps looks enough like a program to be straightforwardly implemented.
- **P2 –** Practice exercises and examples are presented to students by first describing the requirements in plain English. The solution, a pseudo-code or program, is then either provided as is or discussed line-per-line in what is often referred to as a "code walkthrough". The latter allows for explanations as to why the solution is designed in this manner to be added.

The remaining sections will detail the problems inhered to these pedagogies, discuss how instructors often mitigate them with passive learning approaches, discuss the relevance of pair programming as a supplementary active learning approach, suggest ways to improve pair programming, present preliminary evaluation results on the suggested approach & discuss future work.

2. Problem Statement

The previous section identified **P1** as a way to describe the PTP, and **P2** as a way to guide students in acquiring it. However, we argue that both fail at preventing students from circumventing the intended learning.

First, **P2** mostly supports the practice of programming skills for students who have already developed a programming technique on their own without its help. During such learning activities, students are repeatedly shown what amounts to <problem description, solution> pairs. The instruction therefore focusses on the result of the programming activity, the solution, rather than the process of programming. While skilled students will infer the process from observing its results, the ones already struggling with programming will find this instructional approach less than useful. Even worse, by not explicitly guiding them, activities such as P2 allow, if not invite, these students to find a way to achieve the same results with any process within their grasp. The very students who are already in need of support therefore end up spending hours every week honing the wrong skills, yet still complete assignments with some success.

John Bigg's Constructive Alignment theory [1] reminds educators that assessments, and by extension practice activities, need to align with learning outcomes. In this situation, while P2 does not explicitly require students to work in a manner not aligned with the goal of teaching them how to program, it doesn't prevent them from leveraging alternative process which might be harmful to their learning.

An example of such an alternative process has already been reported elsewhere [2]. By presenting students with <problem description, solution> pairs, we offer the illusion that programming might be reduced to a pattern matching problem. When confronted with a new problem, students who didn't develop a PTP on their own will literally start by looking through their slides, lectures notes or textbook for a similar problem. When one is found, its solution is cut and pasted. In the best scenarios, the student then starts modifying it to fit the new problem description. However, because he or she lacks an internal model of the PTP to start off with, these modifications might take on an almost random nature. The original solution is more or less randomly modified then the new version is tested to see whether it behaves closer to the expected behavior.

It might seem obvious that the constructive alignment of an introduction to programming might be re-established through **P1** in so far that it is expected to expose explicitly the PTP. However, we argue that approaches such as **P1** present an unrealistic model of programming.

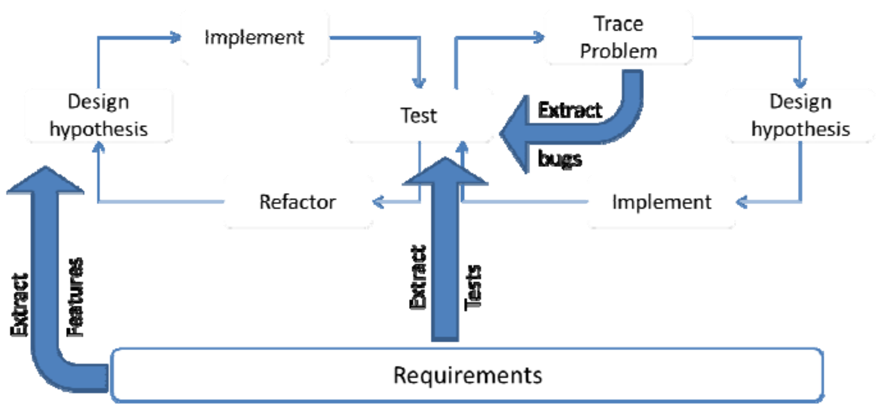
P1 presents an ideal model whereas most students will develop programs in a more iterative manner, making hypotheses, implementing them, troubleshooting bugs, learning from them, adjusting accordingly their initial attempts. Students engaged in such a PTP are on the right track, yet they will be discouraged and feel inadequate if they compare their approaches to the ideal model being lectured. This by itself might shed some light on the attrition rates observed nation-wide in introductions to programming. However, the "ideal model" is not only divergent with students' practice, which might be chalked up to a lack of guidance, but also with modern professional practices. Agile developers rely on programming processes which are no longer compatible with the design-then-implement-in-one-attempt model presented by **P1** [3][4].

By presenting an unrealistic process, which is far removed from the trial-and-error approach students will naturally adopt, P1 fails at addressing the need for making explicit the PTP leveraged by students during P2. This leads to a lack of constructive alignment in students' practice but also to students being discouraged by the apparent incompatibilities between their attempts at thinking through programming problems and the ideal model from the lectures. The lack of scaffolding to guide students from pure trial-and-error to something more in line with agile practices, might account for the nation-wide attrition rates observed in introductions to programming [5][6] or even their negative impact on students' self-efficacy [7].

This situation has also the potential to damage not only students' learning but also instructors and researchers efforts to improve teaching methods meant for novice programmers. Innovations such as leveraging testing to provide students with feedback [8] end up exacerbating the issue of random programming; we make it easy for students to "shake the box" until it works. Further to this, since P1 failed to provide appropriate guidance, we find ourselves in a situation similar to that described by [9][10] whereby any effort to leverage constructivist teaching methods might be hindered from the start.

3. Existing Solutions

It is possible to mitigate the problems detailed in the previous section by using passive learning pedagogies of contents. The keystone is to explicitly expose the PTP to students. The first step in doing so is easily achieved by supplementing lectures with an effort to formalize the PTP used when a programmer develops a solution based on a never seen before problem description. The essential pitfall to avoid is to replicate an "ideal" depiction of the PTP which, as does P1, might both mislead and discourage novices. The authors present the PTP as illustrated in the following diagram. This approach highlights the iterative nature of programming along with its relationship to debugging and testing. As such, it re-establishes the lost constructive alignment by refocusing the teaching efforts on the very skill we expect students to develop. It also contributes to help students understand the relation between programming and other applications of scientific thinking. Hypotheses are replaced by code assumed to be able to fulfill a requirement or implement a feature, experiments are tests designed to refute the hypotheses.



The second step also involves passive learning but supplements the above lecture-style discussions with a "live coding" approach [3]. Such approach allows the instructor to leverage cognitive apprenticeship [11] in order to impart a model of the very thought processes they are expected to replicate to students. The cognitive apprenticeship model is particularly well suited to expose the real nature of the PTP; the instructor thinks aloud while identifying the features to implement, design a solution, implements it, test it and debugs "live" the errors which were invariably introduced. Enabling students to witness typos, bugs, on the fly improvements, also provides them with an explicit guidance on the very skill we expect them to develop. However, it also provides them with a realistic, instead of "ideal", model to which they might better relate. For most students, this realism might go a long way toward motivating them; solutions are not expected to be obtained instantaneously.

The above two interventions might be sufficient to supplement traditional programming pedagogies in order to teach the PTP itself. However, both traditional lecturing and "live coding" rely on an instructivist pedagogy. Students are shown the behavior to model very much like they were previously shown the solutions to obtain. While we successfully addressed the re-focus on showing the right skills, this approach might be improved by exploring the introduction of some constructivist elements in it. While education researchers are still debating the benefits of constructivism [9][10], many computing education researchers have reported success in leveraging constructivist pedagogies with novice programmers [12]. The consensus seems to be that absolute novices need some degree of guidance before being able to fully benefit from active learning pedagogies. Since we provide this initial guidance in the previously mentioned passive learning intervention, the ground is prepared for the introduction of active learning.

Pair programming has been reported as an excessively successful approach in introductions to programming [13]. However, pair programming has initially been designed as a practice for professional developers. As such, it is not meant to address typical educational problems such as the disparity of expertise between students in a pair or whether they instruct one another in an instructivist or constructivist manner. It is reasonable to expect for most students to be focused solely on getting the assignment done while some might also be interested in learning proper techniques while doing so. However, it might be unrealistic to expect students to be committed to help their partners acquire skills through an active learning approach. Showing “how it’s done” while the partner is looking, or even worse simply fixing the bug, is expected to be the dominant mode of interaction. Balanced pairs might feature students taking turns “leading” while unbalanced ones might simply offer a “free ride” to one of the students. In educational terms, this didactic represents a step back to instructivism but also to an obfuscation of the PTP. The “observer” in the pair programming team is left to piece together how their “driver” just came to his or her conclusions. While this is not an issue with professional developers, students pairs with disparate skill levels show that, despite the success of pair programming, we might yet still be able to improve it by adapting it further to educational needs. In this instance, the idea would be to prevent some specific behaviors among students engaged in pair programming so as to enable the non-leading partner to focus on learning the PTP without opportunities to revert to a less effective strategy.

4. Improving Pair Programming

While professional programmers have already developed the skills necessary for them to tackle individually the problems they are working on, this is not true of students. It is therefore reasonable to expect a practice developed for professionals, such as pair programming, to benefit from some adaptations when applied in an educational setting. So far the literature has reveal tremendous benefits to leveraging pair programing in introductory settings. However, it is difficult to assess whether the students most enthused by it are only appreciating the opportunity to pass programming assignments by relying on more skilled partners.

Therefore, our first suggestion to improve pair programming is to let students work individually on their assignments prior to allowing them to exchange information. In this variant, students are presented with a problem description and instructed to develop individually both a solution and a test suite to validate it. While some pedagogies rely on developing tests afterward or before the solution, test driven development [14], we have a neutral stance on this issue and try to invite students to develop their solution in parallel with the tests. Because both students are expected to turn in and be graded on individual solutions, there is no possibility for the “get it done” effect to lead one student to bypass the learning needs of the other. Each of them will have to understand their own solution, their flaws and how to address them.

During the second phase of the assignment, students are allowed to exchange information. Our goal is to prevent one student from solving the other student’s errors while still allowing both to exchange meaningful information, helping each of them improve their programs. The solution we devised relies on our usage of test suites. We allow students to exchange test suites with one another while preventing them to look at each other’s programs. The exchange might be done by posting the tests files on a forum, or with dedicated software.

In all but the most trivial assignments, applying another student’s tests on their solution allows students to identify missing features, misinterpreted requirements, logical errors or suboptimal code coverage in their own tests. However, unlike what would happen in a regular pair programming setting, each student is only pointed toward the flaw in their program but never offered a direct solution he or she may apply blindly. The responsibility of generating a hypothesis regarding which part of the program is responsible for the failed tests and how to remedy the issue is still individual.

Therefore this approach addresses the need for ensuring practice assignments do not offer students the possibility to complete them without exercising the very skills we are teaching them.

5. Preliminary Design & Validation

We implemented this type of active learning activity in COP 2510 Programming Concepts and COP 3515 IT Program Design. The former is a Java-based introduction to computing for various majors while the latter is a follow-up taught to Information Technology majors using the C language to strengthen their skills and expose them to low-level concepts (program stack, heap…) in preparation for system-oriented senior-level courses (e.g. operating systems). We used the following exercise as part of a series of early active learning activities using the Raptor flowchart interpreter [15]. In COP 2510, the pace was slower (~4 weeks) since these were meant for absolute neophytes. In COP 3515, they were used as review material (1-2 weeks maximum).

The exercise was 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 extensively test your program. This time you will write down the tests you have been performing.

A table was provided for students to design and record their tests. This table had 4 columns labeled "Value for TEMP", "Value for SUMMER", "Expected outcome" and "Observed outcome".

Finally, the next step extended the testing to the student's neighbor's code. At this point the objective was clearly to "attack" each other's code through the test harness in order to reveal its flaws.

Now that you have 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 any more bugs.

After we used this exercise, an anonymous online survey evaluated students' attitude about its usefulness. We used a 5 points Likert scale question: "Indicate your level of agreement with the following statement: The flowchart-testing exercises were useful". The response scale offered the usual "strongly disagree" to "strongly agree" options. The responses in terms of percentage and number of respondents were as follows;

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
COP2510	10% (1)	10% (1)	30% (3)	30% (3)	20% (2)
COP3515	0%	0%	0%	80% (4)	20% (1)

These results are an early evaluation attempt which only illustrates the potential reaction from students. While further study is necessary to obtain statistical significance, we leveraged a few observations to inform the next step of our action research. Students with preliminary programming experience better appreciated these exercises than absolute beginners. Testing might have been perceived as distracting by novice programmers who were still trying to acquire the fundamentals. Also, experienced students are more likely to welcome new methods or concepts while a portion of absolute beginners is simply put off by programming in itself. It is impossible to determine whether such students' dislike of our activities is due to the activities themselves or their overall aversion to programming.

From the instructor's perspective, this type of activity allowed the early introduction of code testing practices, which can be positive given their usage in corporate settings but also helped the instructor to address common misconceptions which novice programmers have about program correctness [16].

6. Discussion & Future work

This paper discussed how pair programming is both essential to help students acquire programming skills but also flawed in so far that it allows students to bypass the pedagogical intent of practice activities. We proposed to adapt pair programming to address this lack of constructive alignment by having students work individually before collaborating, develop test suites in addition to their solutions, and exchange only these test suites during their collaboration phase. A first experiment using an exercise designed to support this variant led to positive feedback from students. Further evaluation with larger enrollment figures will be our next step in assessing the impact of our method. Similarly, we plan on investigating further variant addressing the potential loss of intentionality which our variant shares with any other pedagogy relying on tests to provide feedback to students.

7. Acknowledgments

This material is based in part upon work supported by the National Science Foundation under award number DUE CCLI 0836863. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. References

[1] John Biggs (1999): Teaching for Quality Learning at University, (SRHE and Open University Press, Buckingham)

[2] Gaspar, A., Langevin, S. (2007b), Restoring "Coding With Intention" in Introductory Programming Courses, SIGITE 2007, *International conference of the ACM Special Interest Group in Information Technology Education*, July 12-15, Orlando, FL

[3] Kent Beck, Cynthia Andres , Extreme Programming Explained: Embrace Change, 2nd Edition, paperback, Publication Date: November 26, 2004, ISBN-10: 0321278658, ISBN-13: 978-0321278654, Addison Wesley

[4] Ken Schwaver, Mike Beedle, (2001), Agile Software Development with Scrum, ISBN-10: 0130676349, Prentice Hall

[5] Sheard, J. and Hagan, D. (1998): Our failing students: a study of a repeat group. ACM SIGCSE Bulletin 30(3): 223-227.

[6] Robins, A., Rountree, J. and Rountree, N. (2003): Learning and teaching programming: a review and discussion. Journal of Computer Science Education 13(2): 137-172.

[7] A. Gaspar, S. Langevin, N. Boyer, W. Armitage, 2009, Self-Perceived and Observable Self-Direction in an Online Asynchronous Programming Course using Peer Learning Forums. JCSE, Journal of Computer Science Education, Vol. 19, No. 4, pp. 233-255, Special issue on web-based technologies for social learning in computer science education. J. Finlay editor. December 2009.

[8] Nick Parlante, Javabat web application, Standford, http://codingbat.com/, last accessed 4/2012

[9] Mayer, R. (2004). "Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction". American Psychologist 59 (1): 14–19. doi:10.1037/0003-066X.59.1.14. PMID 14736316.

[10] Kirschner, P. A.; Sweller, J.; Clark, R. E. (2006). "Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching". Educational Psychologist 41 (2): 75–86.

[11] Collins, A., Brown, J. S., & Newman, S. E. (1987). Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics (Tech Report No. 403). BBN Laboratories, Cambridge, MA. Centre for the Study of Reading, University of Illinois. January, 1987

[12] Tom Wulf. 2005. Constructivist approaches for teaching computer programming. In Proceedings of the 6th conference on Information technology education (SIGITE '05). ACM, New York, NY, USA, 245-248.

[13] Grant Braught, Tim Wahls, and L. Marlin Eby. 2011. The Case for Pair Programming in the Computer Science Classroom. Trans. Comput. Educ. 11, 1, Article 2 (February 2011), 21 pages. DOI=1921607.1921609 http://doi.acm.org/1921607.1921609

[14] Langr, J. (2005), *Agile Java: Crafting code with Test Driven Development*, Pearson publisher

[15] Carlisle, M.C., Wilson, T.A., Humphries, J. W., Hadfield, S.N. (2005), Raptor: a visual programming environment for teaching algorithmic problem solving, *36th SIGCSE technical symposium on CS education*, St Louis, Missouri, USA, 2005, pp. 176-180

[16] Yifat Ben-David Kolikant. 2005. Students' alternative standards for correctness. In Proceedings of the first international workshop on Computing education research (ICER '05). ACM, New York, NY, USA, 37-43.