Category: Human Aspects of Technology

Constructivist Apprenticeship through Antagonistic Programming Activities

Alessio Gaspar
University of South Florida, Lakeland, USA

Sarah Langevin
University of South Florida, Lakeland, USA

Naomi Boyer
University of South Florida, Lakeland, USA

INTRODUCTION

Computer programming involves more than thinking of a design and typing the code to implement it. While coding, professional programmers are actively on the lookout for syntactical glitches, logic flaws, and potential interactions of their code with the rest of the project. Debugging and programming are therefore not to be seen (and taught) as two distinct skills, but rather as two intimately entwined cognitive processes. From this perspective, teaching programming requires instructors to also teach students how to read code rigorously and critically, how to reflect on its correctness appropriately, and how to identify errors and fix them.

Recent studies indicate that those students who have difficulties in programming courses often end up coding without intention (Gaspar & Langevin, 2007). They search for solved exercises whose descriptions are similar to that of the new problem at hand, cut and paste their solutions, and randomly modify the code until it compiles and passes the instructor's test harness. This behavior is further exacerbated by textbooks, which only require students to modify existing code, thus ignoring the creative side of programming. Breaking this cognitive pattern means engaging students in activities that develop their critical thinking along with their understanding of code and its meaning.

This article discusses constructivist programming activities that can be used in undergraduate programming courses at both the introductory and intermediate levels in order to help students acquire the necessary skills to read, write, debug, and evaluate code for correctness. Our constructivist apprenticeship approach builds on earlier field-tested apprenticeship models of programming instruction that successfully address the learning barriers of the new generations of novice programmers. We go one step further by realigning such approaches to the genuine difficulty encountered by students in a given course, while also addressing some pedagogical shortcomings of the traditional apprenticeship instructional practice. This is achieved by introducing a strong pedagogical constructivist component at the instructional level through so called antagonistic programming activities (APA). We conclude with a manifesto for a new multidisciplinary research agenda that merges the perspectives on learning found in both the computing education and evolutionary computation research communities.

BACKGROUND

Novice Programmers and their Learning Barriers

The study of the learning barriers encountered by novice programmers is critical to the computing education research community. Recent studies describing the misconceptions and preconceived notions held by novice programmers (Chen, Lewandowski, McCartney, Sanders, & Simon, 2007; Kolikant, 2005) indicate that these learning barriers evolve with each new generation of students. In this context, a phenomenon known as "programming without intention" has been identified as an attempt by students who encounter difficulties in programming to mechanize the programming thought process. Their heuristic boils down to the following: (a) reading the description of the program to write and look up available documentation (solved exercises, Google, Krugle, etc.) for another similar, already-solved exercise, (b) cutting and pasting the solution to that exercise as a starting point for the current assignment, and (c) compiling and running the program and, since it most likely does not do what is expected, modifying it. Due to the lack of understanding of the solution being reused and the lack of time devoted to understand the programming activity from the ground up (e.g., learn the syntax, learn the role of statements, learn when to use which), these modifications often boil down to a series of almost random changes until the program seems to execute according to the requirements.

This obviously random-based development approach has very little to do with programming and leaves students unable to explain why a particular statement is in their code. In some occurrences, students stated, "I have the code now for this assignment; I need to understand it." This indicates a complete reversion of the programming thought process leading from ideas to implementations. Instead, intentionality is lost, and statements are manipulated in an almost mechanical manner without second thoughts. Essentially, students are utilizing skills at the lower end of the knowledge framework by demonstrating cognitive functions that Bloom (1956) would have termed as knowledge or understanding with no ability to analyze, synthesize, or evaluate the programming process itself.

Criticizing this approach is, however, insufficient. Understanding what reinforces our students' belief that they are problem solving when developing code this way is what can really help us lead them to overcome this particular learning barrier. The nature of the exercises typically found in some introductory programming courses might be partly responsible for this situation. Often, novice programmers are only required to reuse already working programs and modify them slightly (under heavy guidance) to do something new. While analogical thinking is essential to the professional developer when learning new languages, technologies, and paradigms, it is not safe for it to be the only conceptual tool developed by students during their first programming experience. Creative thinking, critical thinking (e.g., debugging), and problem solving are all essential components of the programming thought process, which, if not given proper attention from the beginning, might fuel the misconception that programming is just a matter of pattern matching in a big book of existing solutions.

Leveraging Apprenticeship in Programming Courses

This learning barrier can be addressed by an apprenticeship model of teaching (Kolling & Barnes, 2004), which can take on several distinct forms. The most obvious one is instructor-led live coding: An instructor presents a problem to her or his students, lets them work on it for a definite time, and then introduces the solution. Instead of presenting students with a detailed explanation of the complete solution, the instructor builds the solution from scratch in front of his or her audience. This diverges from the usual instructional pattern, which leads students to build a dictionary of problem-solution pairs that were introduced in class. Such courses encourage students to memorize data in the hopes that they will be able to simply regurgitate it at the next exam. If a question dares differ from a previously solved problem in any significant way, they will then attempt to fit the memorized solution to this new problem by applying a couple of minor adjustments, which could be stumbled upon almost randomly. By developing the solution in front of the students, the instructor's teaching is aligned with the learning outcomes of the course: the programming thought process itself vs. its outcomes. This approach is clearly illustrated in the work of the BlueJ team and their textbook (Kolling & Barnes, 2004).

Other implementations of the apprenticeship model of teaching are closer to problem-based learning approaches; students are taught the programming thought process by applying it frequently to solve new problems from scratch. This learn-by-programming or learn-by-doing approach also leads students to realize the importance of creative and critical thinking in the programming activity while reducing the benefits of memorization-only or analogy-only strategies. In complement, these pedagogical strategies are often coupled with peer learning approaches (McDowell, Hanks, & Werner, 2003; Willis, Finkel, Gennet, & Ward, 1994).

These apprenticeship pedagogical strategies address the above-mentioned learning barriers by aligning the skills being practiced by students during exercise sessions with the authentic learning outcomes expected from an introductory programming course. This in itself complements nicely with constructive alignment theory (Biggs, 2003), which aligns assessment tools with expected learning outcomes.

From Apprenticeship to Constructivist Apprenticeship

Despite these significant pedagogical achievements, the apprenticeship model of instruction can be further improved from the instructional method perspective. Let us take a critical look at the above-mentioned apprenticeship activities: instructors demonstrating the programming thought process while solving a problem live, classmates developing code while other students play the role of a peer programming observer, students coding against each other in a game-based learning environment (e.g., Bierre, Ventura, Phelps, & Egert, 2006).

These activities are essentially instructivist in nature; students are presented with a problem, they work on it, and then the instructor (or their peer) corrects them or even develops a complete solution for them. Even though the thought process is the focus of the demonstration rather than the solution itself, the teaching process is mostly unilateral. The "sage on the stage" (or next seat) strikes again and leads students to adopt a rather passive attitude as they receive their instruction.

Besides the motivational or attention-span issues that such approaches can cause, the work invested by students to develop their own solution is completely ignored in the instructional process (a hallmark of instructivist pedagogies). They are therefore never corrected, improved, or even