Constructionism and De-Constructionism: Opposite yet Complementary Pedagogies

Jean M. Griffin • Temple University, USA • jeaniacgriffin/at/gmail.com

> Context • Constructionism, Papert's pedagogy and learning theory, involves experiential learning where students engage in exploration, create things that are personally meaningful, and share them with others. This approach is quite motivating, evidenced by the popularity of maker spaces, hackathons, and educational technologies that promote creative computing. With constructionism, the learner's choice is important. This means that learning is often serendipitous. It also means that people often abandon their designs when obstacles arise. This is problematic in learning environments where coverage of key concepts is necessary, practice to develop skills is essential, and persistence with troubleshooting errors is required. **> Problem** • How can teachers and instructional designers complement a constructionist approach with one that addresses its limitations? I introduce de-constructionism, a pedagogy and learning theory that emphasizes learning from taking things apart. It is inspired by reverse engineering, cognitive load theory, practice theory, and theories of learning from errors and negative knowledge. This approach is applicable to computer science, as described here, and other disciplines. **> Method** • I report on a design-based research experiment, where university students interacted with Python practice problems during weekly labs. The designs of the individual problems, and series of problem sets, were based on a model for de-construction developed by the author. **> Results** • The experiment serves as a successful proof of concept for implementing practice problems designed with a de-constructionist approach. Few technical difficulties arose, and the students enjoyed the learning experience. A few revisions to the model for de-construction were warranted. **> Implications** • I provide teachers and instructional designers with a simple, practical way to think about instruction, in terms of construction and de-construction. The de-constructionist approach involves ample, effective practice with taking apart well-built examples, some with intentional bugs. **> Constructivist content** • I propose a pedagogy that is opposite yet complementary to constructionism. **> Key words** • **Constructionism, de-constructionism, bugs, intentional bugs.**

Introduction

« 1 » Constructionism is a pedagogy and learning theory conceived by Seymour Papert who described it as follows:

> "Constructionism is a synthesis of the constructivist theory of developmental psychology [Piaget's theory], and the opportunities offered by technology to base education for science and mathematics on activities in which students work towards the construction of an intelligible entity rather than on the acquisition of knowledge and facts without a context in which they can be immediately used and understood. A central feature of constructionism is that it goes beyond what is usually called 'the cognitive' to include social and affective facets of mathematics and science education." (Papert 1987: 8)

« 2 » Constructionism enjoyed a wave of popularity in the late 1900s when numerous education initiatives engaged young children with the Logo programming language to explore computing, math, animation, storytelling, and other topics. By the early 2000s Logo's popularity had waned but new constructionist technologies emerged that attracted an even larger number of participants from a wider age range. Examples include Scratch, MIT App Inventor, Lilypad Arduino, and littleBits.

« 3 » There are a few well-known college-level courses that use constructionist technologies and approaches for teaching math (Abelson & DiSessa 1992) and programming (Guzdial 2003; Malan & Leitner 2007; "Mobile CSP," http://mobile-csp.org; and "The Beauty and Joy of Computing," http://bjc.berkeley.edu). Despite such examples, constructionism is often perceived as a pre-college "just for fun" pedagogy. Its emphasis on tinkering and *bricolage* is considered insufficient for professional or pre-professional computer science education (Ben-Ari 1998). Even though Papert valued the process of learning from taking things apart (he delighting in deconstructing gear mechanisms as a child), and even though constructionists sometimes promote learning from taking apart (e.g., via remixing), it is typically not employed systematically.

« 4 » It would be helpful if teachers of traditional CS courses with a problem-solving focus knew about constructionism – its history, guiding principles, and varied techniques (for designing, managing, and evaluating constructionist projects) – be-

cause students of all ages are motivated by making their own computational creations. Although there is a constructionist component to the new American advanced placement (AP) *CS Principles* course, which requires student to design and create a computational artifact, teachers are typically not informed that constructionism is a pedagogy relevant to the course ("AP Computer Science Principles,"[1] Kick & Trees 2015). For teachers who already take a constructionist approach, it would be helpful to know how to counter-balance its typical focus on design (Kafai & Resnick 1996)[2] with a pedagogy that explicitly focuses on analysis, skill building, and troubleshooting. The Background section of this target article outlines the theoretical framework for such a pedagogy: de-constructionism. The Method and Results sections describe an experiment with a university Python programming course, where the lab exercises were designed with a de-constructionist approach. The Discussion section considers the experiment within a constructionist/de-constructionist dialectic. It discusses future work, suggests ways to incorporate student collaboration, and invites others to apply de-constructionism to topics other than programming.

Background

« 5 » The Background section discusses guiding principles for de-constructionism: learning from taking apart well-built examples, learning from taking apart well-built examples with intentional errors, and learning through effective practice. Note that de-constructionism is used here to mean a pedagogy; this is distinct from uses of the term deconstruction relative to literary analysis, social theory, architecture, or as a general decomposition strategy (e.g., Boytchev 2015; Self 1997).

Learning from taking apart well-built examples

« 6 » How do people become mechanical engineers? Car mechanics? Fashion designers? In disciplines that involve physical objects, there is a long tradition of learning-by-taking-apart. Future mechanics and mechanical engineers are often drawn to take apart and fix cars, appliances, gadgets, and toys. Future fashion designers often take apart and then copy others' garment designs while developing their own unique style. While constructionism encourages this practice to some extent through remixing, a remixing experience (which involves changing an existing artifact) is often serendipitous. Students may choose to remix only an object's surface features, and avoid or be unaware of its more complex features.

« 7 » This section reviews several learning-by-taking-apart pedagogies. Two are used for hands-on engineering education: mechanical dissection and Tod Phod Jod. Another technique, worked examples, has been researched extensively for mathematics education and to some extent for computing education. A few additional techniques used in computing education are discussed. These approaches are compared and contrasted to find best practices applicable to a general pedagogy of de-construction.

Mechanical dissection

« 8 » In job training and vocational education settings, it is common for apprentices to disassemble, analyze, and assemble (DAA) vehicles, appliances, and other machines. This is useful for developing not only mechanics, but mechanical engineers and designers (Seabrook 2010; Wu 2008). For college/university mechanical engineering departments, this presents challenges in terms of workspaces, staffing, and storage. Thus, many offer introductory courses that are theoretical. An exception is a Stanford University course that implements a hands-on DAA approach called mechanical dissection (Sheppard 1992).

« 9 » Mechanical dissection is similar to the medical school practice of dissecting cadavers, but instead, students disassemble (and re-assemble) mechanical things such as bicycles, fishing reels, and drills. Later courses engage students with design activities. Guiding questions for the mechanical dissection course are: "How did others solve a particular problem?" and "Why does the solution work?" An important goal is for students to become familiar with the terminology of mechanical engineering. This is accomplished by reading instructions, conversing in a structured team setting, answering questions, and writing reflections. Students are guided to study product diagrams, and to notice labels and categories of parts and systems. Students also explain what they learned, and generate their own labels and categories in written reports and reflections. Names, labels, and categories all serve as metacognitive cues (Bransford et al. 2000). Researchers who observed participants of a Toy Dissection module report "We observed most groups generating a causal chain, in which they traced how a sequence of events propagates from one part of the device to the next" (Roschelle & Linde 1996).

« 10 » Some mechanical dissection activities have students analyze multiple items from the same product family, e.g., coffee machines made by different companies. Students compare them and evaluate design tradeoffs. Comparing and generalizing help students learn design templates, develop design strategies, and think abstractly. Sheppard says: "The reality is that very little design is actually new design. Very good designers have this catalog in their brains of stuff – of mechanisms, of devices, of machine elements" (Wu 2008: 59). Over thirty institutions have implemented mechanical dissection (Agogino, Sheppard & Oladipupo 1992; Regan & Sheppard 1996; Roschelle & Linde 1996; Wood & Agogino 1996). It has been adapted to teach about products that are not solely mechanical; this is called product dissection.

Tod Phod Jod

« 11 » An innovative approach to hands-on engineering education in India is *Tod Phod Jod*. It is similar to mechanical dissection in many respects but offered to children through a series of out-of-school workshops, each several hours long. A rough translation of Tod Phod Jod is "break and make." In the first phase, middle-school-aged boys and girls deconstruct and reconstruct objects such as ceiling fans, clocks, and irons. Later they take apart more complicated things and fashion new things (Jods). Like mechanical

1| https://apcentral.collegeboard.org/courses/ap-computer-science-principles

2| See also "A glimpse into the playful world of Seymour Papert" by Idit Harel, EdSurge, 2016. https://www.edsurge.com/news/2016-08-03-a-glimpse-into-the-playful-world-of-seymour-papert

dissection, Tod Phod Jod encourages students to work hands-on with one or more teammates, and ask questions such as: "How does it work?," "What's inside?," "Who made this?," "Why?," "How?" Also similar is the anticipation that discoveries participants make during Tod Phod (deconstruction) will prove useful later on for Jod (design). Yet another similarity is the goal of learning relevant terminology, by "deconstructing the scientific jargon that is usually learned by rote" (Vishnoi 2012). Tod Phod Jod builds on the lower levels of Bloom's taxonomy (remembering and understanding) to engage youth with higher levels – applying, analyzing, evaluating, and creating ("Tod Fod Jod: To Encourage Students to Discover, Experiment, Innovate"[3]). Unlike mechanical dissection, Tod Phod Jod participants do not fill out detailed worksheets and reports. Instead the emphasis is on having a fun and intellectually stimulating experience. India's national initiative to promote youth interest in STEM education recommends Tod Phod Jod for hands-on, activity-based learning ("Rashtriya Avishkar Abhiyan"[4]).

« 12 » Both mechanical dissection and Tod Phod Jod promote learning through reverse engineering, given this definition:

> "Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." (Chikofsky & Cross 1990: 15)

Both help students develop mental abstract representations, which can later be drawn upon for design. There are other approaches for learning by taking apart physical things, e.g., un-crafting (Murer, Fuchsberger & Tscheligi 2017).

Worked examples

« 13 » Another approach to learning from analyzing things that are well-made is the use of worked examples. John Sweller introduced this pedagogy, where students learn from studying problems that are completely worked out as an expert would solve them (Sweller 1988). Contrary to the prevailing belief that students should spend a lot of time solving problems, Sweller's research showed that learning can be more efficient if students study worked examples while gradually learning to solve problems on their own. This is known as the worked example effect (Sweller 2006). According to cognitive load theory, which Sweller developed as the theoretical foundation for this research, humans can learn and store many schemas (patterns of knowledge) in long-term memory, but have a quite limited amount of working memory that can make sense of new information. Learning can be efficient if the cognitive load is managed and the learner does not experience cognitive overload due to excessive or confusing information. Scaffolding (instructional support) is needed to manage cognitive load. Worked examples can provide such support (Sweller 1988; Sweller & Cooper 1985).

« 14 » Mathematics education researchers have conducted numerous large-scale experiments with worked examples to measure students' learning gains. After decades of research, experts recommend that students spend roughly the same amount of time studying worked examples as they do solving problems on their own (Booth et al. 2015). It is important to consider prior knowledge; students with high prior knowledge may lose expertise if required to study too many rudimentary examples (Kalyuga 2007). Empirical research attests to the benefits of several techniques used in conjunction with worked examples. These include self-explanations, where students explain in their own words or choose an explanation (Chi et al. 1994), comparisons (Rittle-Johnson & Star 2007), sub-goal labels (Catrambone 1998), and feedback (Conati & VanLehn 2000). These same techniques show promise with worked examples for computing education. There is recent empirical research on students explaining programs and algorithms (Margulieux et al. 2016; Sudol-DeLyser, Stehlik & Carver 2012), comparing algorithms (Patitsas, Craig & Easterbrook 2013), interacting with sub-goal labels (Margulieux, Catrambone & Guzdial 2016; Morrison, Margulieux & Guzdial 2015), and getting feedback via intelligent tutoring systems (Di Eugenio et al. 2015; Harsley & Morgan 2015; Sudol-DeLyser, Stehlik & Carver 2012) and electronic books (Ericson, Guzdial & Morrison 2015).

Best practices from mechanical dissection, Tod Phod Jod, worked examples, CS education

« 15 » Mechanical dissection, Tod Phod Jod, and worked examples all guide students to deconstruct well-constructed examples. Students explain how the examples work, identify parts, learn terminology, and compare items. Participants of mechanical dissection and Tod Phod Jod get hands-on feedback, while students that interact with worked examples can get computerized feedback. All of these pedagogies help students learn exemplary design patterns and problem-solving strategies. This prepares students both intellectually and psychologically to master design challenges and problem-solving challenges.

« 16 » A few additional learning-by-taking-apart techniques that are used for teaching programming are noteworthy. These emphasize code comprehension, with the philosophy that it is important that students learn to understand code before, or as, they learn to write code on their own. This stands in contrast to the typical emphasis on code writing. There is evidence that students who are unable to understand code samples are unable to write similar ones accurately (Lister et al. 2004; Lopez et al. 2008). Some researchers investigate the cognitive processes involved in code comprehension (Schulte et al. 2010), while others experiment with curricula designed to promote code comprehension. A variety of curricular approaches engage students with analyzing and comparing well-written code samples (Astrachan & Reed 1995; Deimel & Moffat 1982; Kimura 1979; Linn & Clancy 1992). Some emphasize the importance of tracing code (Lopez et al. 2008)[5] and understanding the invisible/notional machine of a programming language with the help of program visualization aids (du Boulay, O'Shea & Monk 1981; Sorva 2013). One popular technique is to ask students to predict what code does. Another is

3 | https://www.slideshare.net/pmpiii/tod-fod-jod

4 | http://mhrd.gov.in/sites/upload_files/mhrd/files/raa/Order_of_RAA_Guidelines.pdf

5 | See also "Teaching programming: Too much doing, not enough understanding" by Quintin Cutts, 2012. https://www.youtube.com/watch?v=Pim4aYfiZiY

to use completion problems; here students are given well-structured code and asked to complete missing sections of it (Deimel & Moffat 1982; Van Merriënboer & Paas 1990). Completion problems help bridge the gap between understanding code and writing code. Thus, predicting and completing are useful activities for de-construction in addition to explaining, comparing, and labeling.

Learning from intentional errors and bugs

« 17 » Given that future bicycle mechanics and fashion designers typically gain mastery by fixing and mending broken things, it follows that an educator could systematically present a collection of broken items to students to provide knowledge of common problems, and practice fixing them. Neither mechanical dissection nor Tod Phod Jod takes this approach, perhaps due to the logistical challenges of doing this with physical objects.

« 18 » Although people can learn from their own mistakes (Duncker 1945), this is often serendipitous. Could learning be more comprehensive and efficient if instruction included intentional errors? This question sparks controversy. Behavioral psychologists view this approach as undesirable because it may introduce or reinforce misconceptions. Several math education researchers who use intentional errors view the behaviorist philosophy as normative, and characterize their own use of them as controversial (Isotani et al. 2011; Tsamir & Tirosh 2005; Tsovaltzi et al. 2010). In contrast, several theories support the idea that intentional errors can promote learning. These include the theories of cognitive dissonance (Festinger 1957, 1962), negative knowledge (Kaess & Zeaman 1960; Oser et al. 2012), impasse-driven learning (VanLehn 1988), learning from errors (Ohlsson 1996), and overlapping waves (Siegler 2002). Over a dozen experience reports and a handful of experiments for math and CS education involve interventions where carefully designed errors or bugs are intentionally placed in otherwise well-built examples to draw the student's attention to a key concept, common error, or common misconception (Griffin 2019).

Learning through effective practice

« 19 » Most of us are familiar with the expression *practice makes perfect*, but what does education research have to say about practice? The testing effect is a well-known phenomenon. Researchers have found that frequent quizzes provide retrieval practice that helps students learn, retain, and transfer knowledge. This is most successful if tests are distributed over time, if they engage learners in effortful recall rather than rote memorization, and if feedback is given (Brown, Roediger & McDaniel 2014; Dunlosky et al. 2013; Roediger & Butler 2011).

« 20 » Some education researchers view practice from a metacognitive perspective. Annemarie Palincsar and Ann Brown (1984) introduced the reciprocal teaching pedagogy, which guides teachers to explicitly teach metacognitive strategies and encourage students to practice these strategies. Mark Singley and John Anderson (1989) researched the role of practice in developing the ability to transfer knowledge from one context to another. Anders Ericsson and colleagues introduced the idea of *deliberate practice* in response to longstanding beliefs that expertise is innate and that experts do not have to practice much. Considering domains such as chess, music, sports, and writing, they found that a lot of time spent on practice does not necessarily lead to progress. In order for practice to be effective it must have certain qualities. Deliberate practice is focused, effortful, and individualized by a teacher or coach to address weaknesses and build strengths. Although deliberate practice is usually not enjoyable, people do it to master a domain (Ericsson & Charness 1994).

« 21 » Many computing teachers implement *blocked practice*. That is, they introduce topic A and students practice topic A. Then they introduce topic B and students practice topic B, and so on. Education psychology researchers have found that distributing practice over time and interleaving topics is superior to blocked practice. It is also helpful to scaffold instruction from concrete to abstract, to balance examples with problem solving, to provide feedback, and to consider students' prior knowledge (Dunlosky et al. 2013; Koedinger, Booth & Klahr 2013). Regarding learning to program, a beginner's knowledge of programming is often fragile (Perkins & Martin 1989). Practice correlates positively with developing programming skills (Douce, Livingstone & Orwell 2005; Macnamara, Hambrick & Oswald 2014; Ventura 2005). The Leeds group recommends that teachers frequently give students practice problems to develop automaticity with basic programming skills (Lister et al. 2004). Kranch studied novice programmers and argues that they need plenty of practice with fundamentals (Kranch 2011). Unfortunately, many beginning students say they lack the time and motivation to practice (Kinnunen & Malmi 2006).

Research questions

« 22 » The Background section discussed the three guiding principles for de-constructionism. The first two principles were introduced in earlier work: learning from taking apart well-built examples and learning from taking apart well-built examples with intentional errors (Griffin, Kaplan & Burke 2012). This article provides theoretical justification for those principles and for a third principle: learning through effective practice. The research questions that guide this study are:

- How can a de-constructionist approach be implemented in a programming class?
- What are students' attitudes about this approach?

Method

« 23 » I conducted a design-based research study. Design-based research involves iterative experimentation in authentic settings such as classrooms; it typically involves a curricular intervention in cooperation with teachers (Brown 1992; Collins 1990).

> "Design experiments have both a pragmatic bent – 'engineering' particular forms of learning – and a theoretical orientation – developing domain-specific theories by systematically studying those forms of learning and the means of supporting them." (Cobb et al. 2003: 9)

I conducted a usability study, two pilot studies, and a semester-long experiment. Here I report on parts of the semester-long experiment, for which I designed web-based de-constructionist practice problems for learning to program with Python.

Context and participants

« 24 » The study was conducted in cooperation with the CS department of an urban public research university in the northeast United States. In 2015 the university had ~28.5k undergraduate students: 51% female; 11% Asian; 13% African American/Black; and 55% White. The study involved Professors Town and Park (pseudonyms). Each taught two sections of an introductory CS1-Python course with 87 undergraduate students. Both CS majors and non-majors attended the course. 15% of the students were female; 24% identified as a (non-Caucasian, non-Asian) racial or ethnic minority. I was the lab instructor and developed the activities for the weekly labs.

« 25 » I developed a problem set for each lab, starting in week 4, and iteratively designed problems for the following week for a total of ten labs. I used the Runestone Interactive e-book system to create web-based practice problems (Ericson, Guzdial & Morrison 2015; Miller & Ranum 2014). Runestone has a unique collection of components that provide interactivity and real-time feedback. In addition to components for multiple choice and free-response questions, several components help students explore code through interactive code reading. The *clickable* component (Figure 1) allows the student to click on one or more sections of code, e.g., to identify key features or bugs. Using the *drag 'n drop* component (Figure 2), the user clicks and drags items from one column to match items in another, e.g., expressions and values. The *codelens* component (Figure 3) provides program visualization as the user steps through code line by line (Cross, Hendrix & Barowski

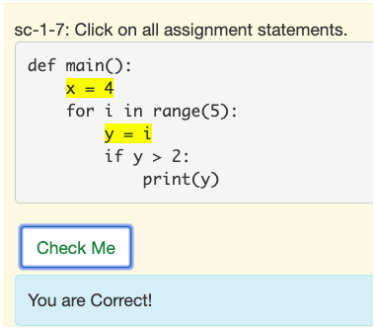


Figure 1 • Clickable component.

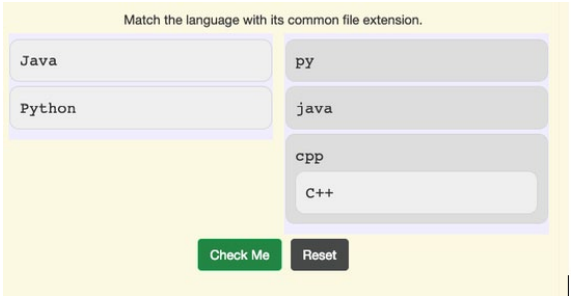


Figure 2 • Drag 'n drop component.

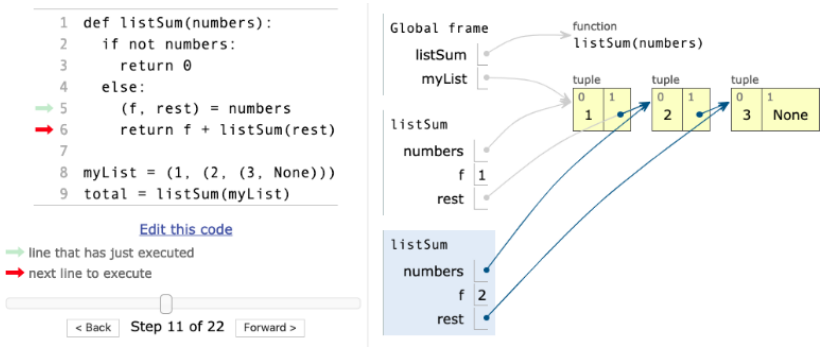
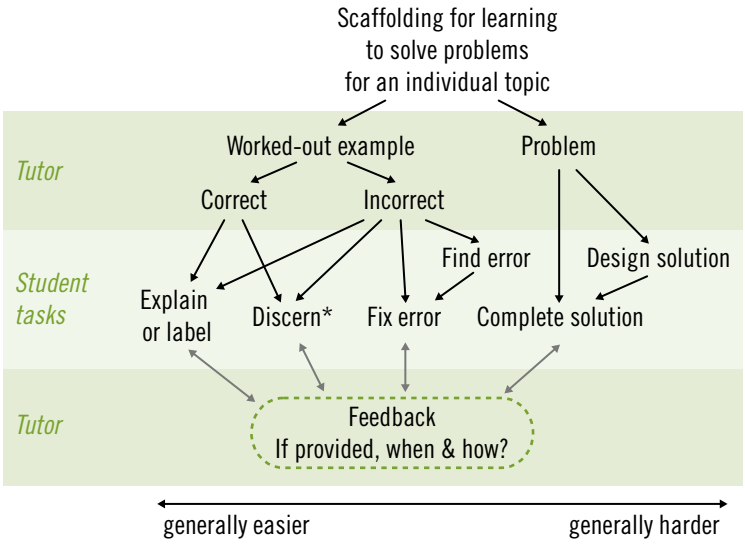


Figure 3 • Codelens component (with online Python tutor).



*categorize, compare, identify, match, pattern match, predict, reverse engineer, unpack, unscramble, etc.

Figure 4 • Model for de-construction.

2011; Guo 2013). With the *Parson's problem* component, the user re-orders scrambled code (Parsons & Haden 2006). Several components require students to write code. The instructional designer may supply starter code (to be completed or changed), or ask the student to write code from scratch. The author may also supply unit tests, to test the user's code and supply feedback. Practice problems related to debugging can be created with any of these components.

« 26 » To guide the instructional design, I used the model for de-construction (Griffin 2016), which I refined during the experiment (Figure 4). I used practice problems that generally progressed from easier ones shown on the left side of the model (e.g., reading and tracing problems that require explaining and labeling worked-out examples) to harder ones shown on the right side of the model that require completing code or writing code from scratch. Most problems involved short code segments. A problem set (one per lab) generally progressed from concrete to abstract. The series of problem sets, which spanned multiple weeks, incorporated distributed practice and interleaving. Students worked on the problems individually but could consult with classmates, with me, or with my teaching assistant. Students who finished before the lab period ended were free to leave. A student who completed all ten labs interacted with 109 practice problems, averaging approximately 11 per lab.

« 27 » The treatment group (Bugs) and control group (NoBugs) were each comprised of two lab sections, one for each teacher. The Bugs group got 22 practice problems with bugs; the NoBugs group got 22 similar reading/tracing problems without bugs. Thus, about 80% of the problems were common, about 20% were specific to condition. Debugging problems were generally introduced as follows. For a given topic, problems with correct example code were introduced first. Next, easy problems with bugs were introduced, such as ones where the bug location was shown, which the student was asked to explain, categorize, or fix. Then more difficult problems were introduced, such as ones that required finding a bug. (These are not hard-and-fast rules.)

Timeline

« 28 » The course spanned a 14-week semester, with two 80-min lectures and one 2-hour lab per week. Class time was devoted to lectures, lab time to practice problems. I met with the professors weekly. They taught the same topics with the same textbook and gave similar exams.

Instruments and procedures

« 29 » At the end of the semester, an online survey was administered to gauge students' attitudes about the practice problems. It had the following Likert-scale questions:

1 | Indicate your level of agreement with the following statement: The "Interactive Python" e-book exercises during lab helped me to learn Python (strongly disagree, disagree, neutral, agree, strongly agree)
2 | In general, I found the e-book exercises to be: (too easy, about the right level of difficulty, too hard)
3 | I recommend that in the future, the labs have: (fewer e-book exercises, about the same number, more)
4 | Did you like the reading/tracing exercises? (strong dislike, dislike, neutral, like, strong like)
5 | Did you like the writing exercises? (strong dislike, dislike, neutral, like, strong like)
6 | Do you think the reading/tracing exercises helped you learn Python? (not helpful, neutral, helpful, very helpful)
7 | Do you think the writing exercises helped you learn Python? (not helpful, neutral, helpful, very helpful)

« 30 » The Bugs group got two additional questions, about liking and learning from the debugging problems. Additional instruments were administered to measure learning gains; related findings are reported elsewhere (Griffin 2019). Throughout the experiment I recorded researcher memos and took field notes about students' interactions with the practice problems in the labs.

Results

« 31 » This section summarizes the responses to the post-attitudes survey.

Attitudes about overall learning, difficulty, and quantity

« 32 » A majority of students responded positively about overall learning, difficulty, and quantity of problems (Table 1). Of the 76 survey respondents, 92% agreed or strongly agreed that the problems helped them to learn Python, 88% thought the level of difficulty was about right, 8% found them too easy, 4% too hard. Regarding dosage, 62% recommended the same amount for future courses, 10% recommended fewer, 28% more.

Attitudes about liking the practice problems

« 33 » A majority of students liked or strongly liked each type of practice problem. Seventy-five students responded to the question about how much they liked the reading/tracing problems. Of these 75 students, 83% liked or strongly liked them, 8% disliked them, 9% were neutral. Of the 76 students who responded to the question about liking the code-writing problems, 87% liked or strongly liked them, 3% disliked them, 8% were neutral. The 39 students in the Bugs group responded about liking the debugging problems, 69% liked or strongly liked them, 5% disliked them, 26% were neutral.

Attitudes about learning from the practice problems

« 34 » A majority of students thought both the reading/tracing and the writing problems helped them to learn, while 38% thought so about the debugging problems. Of the 76 students who answered the question about learning from the reading/tracing problems, 82% thought they were helpful or very helpful, 3% thought they were not helpful, 16% were neutral. About the code-writing problems, 96% of 76 respondents thought they were helpful or very helpful, 1% thought they were not helpful, 3% were neutral. For learning from debugging, an extra response category was added (Detrimental) because the review of the literature suggests that some think errors may deter learning. Of the 39 students in the Bugs group who

Helped me learn Python	Disagree 1%	Not sure 7%	Agree / strongly agree 92%
Level of difficulty	Too hard 4%	Too easy 8%	About right 88%
Amount of practice	Recommend fewer 10%	Recommend more 28%	About the same 62%

Table 1 • Overall attitudes about the practice problems (N=76 respondents).

responded, 38% thought they were helpful or very helpful, 15% thought they were not helpful, one student (3%) thought they were detrimental, 44% were neutral.

Discussion

« 35 » This design-based research study implemented and further developed the emerging pedagogy of de-constructionism. In a 10-week intervention with a quasi-experimental design, undergraduates in a Python programming class solved sets of practice problems designed with a de-constructionist approach. This approach emphasizes ample practice with taking apart well-built examples, some of which have intentional bugs. Before writing code for a given topic, students engaged with interactive reading and tracing code by explaining, comparing, identifying, labeling, matching, tracing, and unscrambling. The treatment group got some problems with intentional bugs.

« 36 » The study sought to understand students' attitudes about learning with a de-constructionist approach. According to the post-attitudes survey, a majority of students liked each type of practice problem (for reading/tracing, writing, and debugging). While a majority of students thought the reading/tracing and writing problems were helpful for learning, only 38% thought so about the debugging problems. Further research is warranted to determine if it is useful to help students see the potential benefits of learning from intentional bugs, such as to help one become aware of common errors and learn how to repair them.

« 37 » As the lab instructor, I observed that students' interactions with the practice problems were generally smooth; they appeared motivated to complete the problem sets. I was assured that all students who attended the lab interacted with the key course concepts. Students appreciated the immediate feedback and the chance to retry until they completed a problem successfully. Approximately a third to half of the students finished during the first hour of the 2-hour lab period. Usually several took the entire period.

« 38 » As the instructional designer, my experience with Runestone Interactive was generally positive. Despite being a fledgling system, it is reliable and responsive, and has a rich variety of components. Creating the practice problems was somewhat laborious. It required encoding them with the reStructuredText language, but Runestone is evolving to streamline this process. I found that implementing distributed practice and interleaved practice was more difficult and time-consuming than implementing blocked practice. Blocked practice is straightforward: teach a new topic, give practice on that topic; repeat. Storing problems for blocked practice is also straightforward – simply organize them by topic. Distributed and interleaved practice involve decisions about when and how to intersperse prior topics with new topics during practice. Decisions about storing them can be complicated, e.g., if the order of topics may change.

« 39 » As the researcher, in some respects it was beneficial for me to also have the roles of lab instructor and instructional designer. This was an early-stage, exploratory study that implemented the Model for de-construction for the first time (Figure 4). The lab exercises were iteratively designed, weekly, based on observations from the previous week; adjustments were made regarding the quantity and difficulty level of the problems, the balance of problems (with respect to reading/tracing, debugging, writing), and choices of Runestone components. In future studies, having two or three people perform these roles would be advantageous.

« 40 » Future plans include incorporating constructionist activities into the lab period. Rather than having students leave when they finish the problem sets, students could work on creative projects. Ideally the rubrics would be such that students who need more time to master the practice problems are not penalized for having less time to work on creative projects. This approach is common with the mastery learning approach (Bloom 1968; Griffin, Pirmann & Gray 2016). Unfortunately, many CS professors are unaccustomed to designing rubrics for creative projects, such as the professors in this study who have doctorates in mathematics and computer science. Guidelines for developing rubrics for creative CS projects are available (Cateté, Snider & Barnes 2016). Introducing constructionism into such learning environments would be valuable for both teachers and students. Students could have motivating design experiences, and teachers could learn how to design, manage, and evaluate creative projects.

« 41 » Additional future plans include adding appealing graphics and gamification elements. Gamification can have positive effects with respect to student motivation and learning outcomes (Hamari, Koivisto & Sarsa 2014). A priority would be to add these elements to the debugging and reading/tracing activities, since students already view writing code as a worthwhile activity. Having students collaborate on de-constructionist activities, e.g., with Pair Programming, is worth exploring. Also desirable would be to differentiate instruction based on prior experience and/or demonstrated mastery.

« 42 » The constructionist/de-constructionist dialectic is useful for teachers and instructional designers, but it is not all-inclusive. Other pedagogies are also important, such as ones that teach problem-solving, or systematic approaches to debugging. While this study explored de-constructionism as it applies to learning programming, it can be implemented for a variety of subjects with physical and/or conceptual elements.

JEAN M. GRIFFIN
is a computer scientist, instructional designer, and education researcher. Griffin taught computer science full-time for ten years at the University of Pennsylvania and part-time for seven years at youth camps, academies, and workshops. After earning a doctorate in education from Temple University she joined Google Research.

Conclusion

« 43 » Constructionism encourages experiential learning, where people create things that are personally meaningful and share them with others. Constructionism is aligned with the constructivist philosophy – as learners build computational artifacts, they build knowledge in the constructivist sense (Papert & Harel 1991). Constructionist learning environments are motivational but insufficient in settings where broad topic coverage, skill-building practice, and essential troubleshooting skills are required. In this article, I introduced de-constructionism as a pedagogy that is opposite yet complementary to constructionism. The de-constructionist approach involves ample, effective practice with taking apart well-built examples, some of which have intentional bugs. The experiment described here serves as a successful proof of concept for implementing practice problems designed with a de-constructionist approach. More research is warranted to discover if and how de-constructionist approaches may be applied alone or in combination with constructionist approaches for teaching a variety of topics.

References

Abelson H. & DiSessa A. (1992) Turtle geometry. MIT Press, Cambridge MA.

Agogino A. M., Sheppard S. & Oladipupo A. (1992) Making connections to engineering during the first two years. In: Proceedings of the twenty-second annual conference Frontiers in education (FIE '92). IEEE Press, Piscataway NJ: 563–569.

Astrachan O. & Reed D. (1995) AAA and CS 1: The applied apprenticeship approach to CS 1. In: Papers of the 26th SIGCSE technical symposium on computer science education (SIGCSE '95). ACM, New York: 1–5.

Ben-Ari M. (1998) Constructivism in computer science education. ACM SIGCSE Bulletin 30(1): 257–261.

Bloom B. S. (1968) Learning for mastery. UCLA Evaluation Comment 1(2): 1–11. https://files.eric.ed.gov/fulltext/ED053419.pdf

Booth J. L., McGinn K. M., Young L. K. & Barbieri C. (2015) Simple practice doesn't always make perfect: Evidence from the worked example effect. Policy Insights from the Behavioral and Brain Sciences 2(1): 24–32.

Boytchev P. (2015) Constructionism and deconstructionism. Constructivist Foundations 10(3): 355–369.
▸ https://constructivist.info/10/3/355

Bransford J. D., Brown A. L., Cocking R. R., Donovan M. S. & Pellegrino J. W. (eds.) (2000) How people learn: Brain, mind, experience, and school. Expanded edition. National Academy Press, Washington DC.

Brown A. L. (1992) Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. Journal of the Learning Sciences 2(2): 141–178.

Brown P. C., Roediger III H. L. & McDaniel M. A. (2014) Make it stick: The science of successful learning. The Belknap Press of Harvard University Press, Cambridge MA.

Cateté V., Snider E. & Barnes T. (2016) Developing a rubric for a creative CS principles lab. In: Proceedings of the 2016 ACM conference on innovation and technology in computer science education (ITiCSE '16). ACM, New York: 290–295.

Catrambone R. (1998) The subgoal learning model: creating better examples so that students can solve novel problems. Journal of Experimental Psychology: General 127(4): 355–376.

Chi M. T. H., De Leeuw N., Chiu M.-H. & Lavancher C. (1994) Eliciting self-explanations improves understanding. Cognitive Science 18(3): 439–477.

Chikofsky E. J. & Cross J. H. (1990) Reverse engineering and design recovery: A taxonomy. IEEE Software 7: 13–17.

Cobb P., Confrey J., DiSessa A., Lehrer R. & Schauble L. (2003) Design experiments in educational research. Educational Researcher 32(1): 9–13.

Collins A. (1990) Towards a design science of education. Technical Report No. 1. Center for Technology in Education, New York.

Conati C. & VanLehn K. (2000) Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. International Journal of Artificial Intelligence in Education 11(1): 389–415.

Cross J. H., Hendrix T. D. & Barowski L. A. (2011) Combining dynamic program viewing and testing in early computing courses. In: Proceedings of the 35th Annual IEEE computer software and applications conference. IEEE Press, Piscataway NJ: 184–192.

Deimel L. E. & Moffat D. V. (1982) A more analytical approach to teaching the introductory programming course. In: J. Smith & M. Schuster (eds.) Proceedings of the NECC. The University of Missouri, Columbia: 114–118.

Di Eugenio B., Green N., Alzoubi O., Alizadeh M., Harsley R. & Fossati D. (2015) Worked-out examples in a computer science intelligent tutoring system. In:

Proceedings of the 16th annual conference on information technology education. ACM, New York: 121.

Douce C., Livingstone D. & Orwell J. (2005) Automatic test-based assessment of programming: A review. ACM Journal of Educational Resources in Computing 5(3): 1–13.

du Boulay B., O'Shea T. & Monk J. (1981) The black box inside the glass box: Presenting computing concepts to novices. International Journal of Man-Machine Studies 14: 237–249.

Duncker K. (1945) On problem-solving. Psychological Monographs 58(5).

Dunlosky J., Rawson K. A., Marsh E. J., Nathan M. J. & Willingham D. T. (2013) Improving students' learning with effective learning techniques: Promising directions from cognitive and educational psychology. Psychological Science in the Public Interest 14(1): 4–58.

Ericson B. J., Guzdial M. J. & Morrison B. B. (2015) Analysis of interactive features designed to enhance learning in an ebook. In: Proceedings of the 11th international conference on computing education research (ICER '15). ACM, New York: 169–178.

Ericsson K. A. & Charness N. (1994) Expert performance: Its structure and acquisition. American Psychologist 49(8): 725–747.

Festinger L. (1957) A theory of cognitive dissonance. Stanford University Press, Stanford CA.

Festinger L. (1962) A theory of cognitive dissonance. Volume 2. Stanford University Press, Stanford CA.

Griffin J. M. (2016) Learning by taking apart: Deconstructing code by reading, tracing, and debugging. In: Proceedings of the 17th annual conference on information technology education (SIGITE'16). ACM, New York: 148–153.

Griffin J. M. (2019) Designing intentional bugs for learning. In: Proceedings of the first UK and Ireland Computing Education Research conference. ACM, Canterbury UK, in press.

Griffin J. M., Kaplan E. & Burke Q. (2012) Debug'ems and other Deconstruction Kits for STEM learning. In: Proceedings of the second IEEE integrated STEM education conference (ISEC '12). IEEE Press, Piscataway NJ: 1–4.

Griffin J. M., Pirmann T. & Gray B. (2016) Two teachers, two perspectives on CS principles. In: Proceedings of the 47th ACM technical symposium on computer science education (SIGCSE '16). ACM, New York: 461–466.

Guo P. J. (2013) Online Python tutor: Embeddable web-based program visualization for CS education. In: Proceeding of the 44th ACM technical symposium on computer science education. ACM, New York: 579–584.

Guzdial M. (2003) A media computation course for non-majors. In: Finkel D. (ed.) Proceedings of the 8th annual conference on innovation and technology in computer science education (ITiCSE '03). ACM, New York: 104–108.

Hamari J., Koivisto J. & Sarsa H. (2014) Does gamification work? A literature review of empirical studies on gamification. In: Proceedings of the 47th Hawaii international conference on system sciences (HICSS '14). IEEE Computer Society, Washington DC: 3025–3034.

Harsley R. & Morgan S. (2015) Learning together: Expanding the one-to-one ITS model for computer science education. In: Proceedings of the eleventh annual international conference on international computing education research (ICER '15). ACM, New York: 263–264.

Isotani S., Adams D., Mayer R. E., Durkin K., Rittle-Johnson B. & McLaren B. M. (2011) Can erroneous examples help middle-school students learn decimals? Proceedings of the Sixth European Conference on Technology Enhanced Learning: Towards Ubiquitous Learning (EC-TEL-2011): 1–14.

Kaess W. & Zeaman D. (1960) Positive and negative knowledge of results on a pressey-type punchboard. Journal of Educational Psychology 60(1): 12–17.

Kafai Y. B. & Resnick M. (1996) Constructionism in practice: Designing, thinking, and learning in a digital world. Routledge, London.

Kalyuga S. (2007) Expertise reversal effect and its implications for learner-tailored instruction. Educational Psychology Review 19(4): 509–539.

Kick R. & Trees F. P. (2015) AP CS principles: Engaging, challenging, and rewarding. ACM Inroads 6(1): 42–45.

Kimura T. (1979) Reading before composition. In: Proceedings of the 10th SIGCSE technical symposium on computer science education (SIGCSE '79). ACM, New York: 162–166.

Kinnunen P. & Malmi L. (2006) Why students drop out CS1 course? In: Proceedings of the second international workshop on computing education research (ICER '06). ACM, New York: 97–108.

Koedinger K. R., Booth J. L. & Klahr D. (2013) Instructional complexity and the science to constrain it. Science 342(6161): 935–937.

Kranch D. A. (2011) Teaching the novice programmer: A study of instructional sequences and perception. Education and Information Technologies 17(3): 291–313.

Linn M. C. & Clancy M. J. (1992) The case for case studies of programming problems. Communications of the ACM 35(3): 121–132.

Lister R., Adams E. S., Fitzgerald S., Fone W., Hamer J., Lindholm M., McCartney R., Moström J. E., Sanders K., Seppälä O., Simon B. & Thomas L. (2004) A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bulletin 36(4): 119–150.

Lopez M., Whalley J., Robbins P. & Lister R. (2008) Relationships between reading, tracing and writing skills in introductory programming. In: Proceedings of the 4th international workshop on computing education research (ICER '08). ACM, New York: 101–112.

Macnamara B. N., Hambrick D. Z. & Oswald F. L. (2014) Deliberate practice and performance in music, games, sports, education, and professions: A meta-analysis. Psychological Science 25(8): 1608–1618.

Malan D. J. & Leitner H. H. (2007) Scratch for budding computer scientists. ACM SIGCSE Bulletin 39(1): 223–227.

Margulieux L. E., Catrambone R. & Guzdial M. (2016) Employing subgoals in computer programming education. Computer Science Education 26(1): 44–67.

Margulieux L. E., Morrison B. B., Guzdial M. & Catrambone R. (2016) Training learners to self-explain: Designing instructions and examples to improve problem solving. In: Looi C. K., Polman J. L., Cress U. & Reimann P. (eds.) Transforming learning, empowering learners: The international conference of the learning sciences (ICLS) 2016, Volume 1. International Society of the Learning Sciences, Singapore: 98–105.

Miller B. & Ranum D. (2014) Runestone interactive: Tools for creating interactive course materials. In: Proceedings of the First ACM

conference on learning @ scale (L@S' 14). ACM, New York: 213–214.

Morrison B. B., Margulieux L. E. & Guzdial M. (2015) Subgoals, context, and worked examples in learning computing problem solving. In: Proceedings of the 11th international conference on computing education research (ICER '15). ACM, New York: 21–29.

Murer M., Fuchsberger V. & Tscheligi M. (2017) Un-crafting: De-constructive engagements with interactive artifacts. In: Proceedings of the ninth international conference on tangible, embedded, and embodied interaction (TEI '17). ACM, New York: 67–77.

Ohlsson S. (1996) Learning from error and the design of task environments. International Journal of Educational Research 25(5): 419–448.

Oser F., Näpflin C., Hofer C. & Aerni P. (2012) Towards a theory of negative knowledge (NK): Almost-mistakes as drivers of episodic memory amplification. In: Bauer J. & Harteis C. (eds.) Human fallibility: The ambiguity of errors for work and learning. Springer, New York: 53–70.

Palincsar A. S. & Brown A. L. (1984) Reciprocal teaching of comprehension monitoring activities. Cognition and Instruction 1: 117–175.

Papert S. (1987) Constructionism: A new opportunity for elementary science education. National Science Foundation proposal. MIT, Cambridge MA.

Papert S. & Harel I. (1991) Situating constructionism. In: Papert S. & Harel I. (eds.) Constructionism. Ablex Publishing, Norwood NJ: 1–11.

Parsons D. & Haden P. (2006) Parson's programming puzzles: A fun and effective learning tool for first programming courses. In: Proceedings of the 8th Australasian conference on computing education. Volume 52. Australian Computer Society, Darlinghurst: 157–163.

Patitsas E., Craig M. & Easterbrook S. (2013) Comparing and contrasting different algorithms leads to increased student learning. In: Proceedings of the 9th international conference on computing education research (ICER '13). ACM, New York: 145–152.

Perkins D. & Martin F. (1989) Fragile knowledge and neglected strategies in novice programmers. In: Soloway E. & Spohrer J. (eds.) Studying the novice programmer. Lawrence Erlbaum Associates, Hillsdale NJ: 213–229.

Regan M. & Sheppard S. (1996) Interactive multimedia courseware and the hands-on learning experience: An assessment study. Journal of Engineering Education 85(2): 123–132.

Rittle-Johnson B. & Star J. R. (2007) Does comparing solution methods facilitate conceptual and procedural knowledge? An experimental study on learning to solve equations. Journal of Educational Psychology 99(3): 561–574.Roediger H. L. & Butler A. C. (2011) The critical role of retrieval practice in long-term retention. Trends in Cognitive Sciences 15(1): 20–27.

Roschelle J. & Linde C. (1996) Toy dissection: Formative in-depth assessment report. Institute for Research on Learning (IRL), Palo Alto CA. http://www-adl.stanford.edu/images/toyasess.pdf

Schulte C., Clear T., Taherkhani A., Busjahn T. & Paterson J. H. (2010) An introduction to program comprehension for computer science educators. In: Proceedings of the 2010 ITiCSE working group reports. ACM: New York: 65–86.

Seabrook J. (2010) How to make it. The New Yorker 20 September 2010. http://www.newyorker.com/magazine/2010/09/20/how-to-make-it

Self J. (1997) From constructionism to deconstructionism: Anticipating trends in educational styles. European Journal of Engineering Education 22(3): 295–307.

Sheppard S. D. (1992) Mechanical dissection: An experience in how things work. In: Proceedings of the conference on engineering education: Curriculum innovation & integration. 6–10 January 1992, Santa Barbara CA. http://www-adl.stanford.edu/images/dissphil.pdf

Siegler R. S. (2002) Microgenetic studies on self-explanation. In: Granott N. & Parziale J. (eds.) Microdevelopment: Transition processes in development and learning. Cambridge University Press, Cambridge UK: 31–58.

Singley M. K. & Anderson J. R. (1989) The transfer of cognitive skill. Harvard University Press, Cambridge MA.

Sorva J. (2013) Notional machines and introductory programming education. ACM Transactions on Computing Education 13(2): 1–31.

Sudol-DeLyser L. A., Stehlik M. & Carver S. (2012) Code comprehension problems as learning events. In: Proceedings of the 17th ACM annual conference on innovation and technology in computer science education (ITiCSE '12). ACM, New York: 81–86.

Sweller J. (1988) Cognitive load during problem solving: Effects on learning. Cognitive Science 12(2): 257–285.

Sweller J. (2006) The worked example effect and human cognition. Learning and Instruction 16(2): 165–169.

Sweller J. & Cooper G. A. (1985) The use of worked examples as a substitute for problem solving in learning algebra. Cognition and Instruction 2(1): 59–89.

Tsamir P. & Tirosh D. (2005) In-service mathematics teachers' views of errors in the classroom. Focus on Learning Problems in Mathematics 27(3): 30.

Tsovaltzi D., Melis E., McLaren B. M., Meyer A., Dietrich M. & Goguadze G. (2010) Learning from erroneous examples: When and how do students benefit from them. In: Wolpers M., Kirschner P. A., Scheffel M., Lindstaedt S. & Dimitrova V. (eds.) Proceedings of the 5th European conference on technology enhanced learning. LNCS 6383. Springer, Heidelberg: 357–373.

Van Merriënboer J. & Paas F. G. W. C. (1990) Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. Computers in Human Behavior 6(3): 273–289.

VanLehn K. (1988) Towards a theory of impasse-driven learning. In: Mandl H. & Lesgold A. (eds.) Learning issues for intelligent tutoring systems. Springer, New York: 19–41.

Ventura Jr. P. R. (2005) Identifying predictors of success for an objects-first CS1. Computer Science Education 15(3): 223–243.

Vishnoi A. (2012) NIC lesson on learning: Tod-Fod-Jod: The Indian Express, New Delhi India. http://archive.indianexpress.com/news/nic-lesson-on-learning-todfodjod/1015042/0

Wood W. H. & Agogino A. M. (1996) Engineering courseware content and delivery: The NEEDS infrastructure for distance-independent education. Journal of the American Society for Information Science 47(11): 863–869.

Wu C. (2008) Some disassembly required. ASEE Prism 18: 56–59.

Received: 21 January 2019
Accepted: 15 April 2019

Open Peer Commentaries

on Jean Griffin's "Constructionism and De-Constructionism"

De-Constructionism: An Effective Premise to Constructionist Learning

Mihaela Sabin
University of New Hampshire, USA
mihaela.sabin/at/unh.edu

> Abstract • As the name suggests, a de-constructionist pedagogy motivated by structured practice to develop skills and troubleshoot errors is opposite to constructionist approaches. And yet the model for de-construction proposed by Griffin affords complementarity to constructionist teaching and learning of introductory computer science at college level. In my response I ask questions about possible model implementations that facilitate complementary constructionist activities naturally progressing from their de-constructionist counterparts.

« 1 » Learning to program is foundational in college-level computer science programs. In Seymour Papert's vision, programming is a means for learning by which students "embark on an exploration about how they themselves think" while they teach "the computer how to think" (Papert 1980: 19). The application of constructionism to learning computer science (CS) with programming tools, however, faces unique disciplinary challenges. In the absence of a preconceived mental model of how a computer works, CS learners are at disadvantage when they try to make viable cognitive structures informed by what they already know. In particular, as Mordechai Ben-Ari points out, "[t]he computer science student is faced with immediate and brutal feedback on conclusions drawn from his or her internal model" (Ben-Ari 1988: 259). To make deeper sense of the feedback given by the computer, CS learners need to construct a viable mental model that helps them predict the computational outcomes of their constructionist artifact – the program they want to write, debug if it has errors to make it work, and then to use it to create more sophisticated programs that implement creative designs.

« 2 » Understanding how to program and achieving programming fluency cannot avoid this essential question: what are the abstract properties of the computing machinery that we control through programming in a particular language? Benedict du Boulay (1989) named these abstractions a "notional machine" to refer to an idealized abstraction of a computer (including computer architecture and characteristics of the runtime environment that executes a program). In contrast to other disciplines where students' experiences with phenomena are a springboard for making sense of things in a personal way, computer science and programming have concepts that are particularly difficult for novice programmers because alternative interpretations misrepresent the notional machine and lead to unproductive and frustrating learning of programming (Ben-Ari 1998; Sorva 2013). What complicates matters is that the status quo of teaching introductory CS is generally missing the importance of the core idea of notional machine, as noted by Juha Sorva (2013):

> "[T]he big picture of how programs work at runtime does not get quite the attention it deserves in the light of learning theory and empirical evidence." (Sorva 2013: 26)

« 3 » Griffin's model for de-construction (§26) uses practice problems that scaffold learning to code from reading and tracing code examples to completing and writing code from scratch. In her design-based research study, students (N = 87) practiced coding during ten weekly 2-hour labs by solving problem sets with problems progressing from concrete to abstract. Students in the treatment group (N = 39) had two lab sections focused on learning from intentional errors and bugs, which were scaffolded from explaining, categorizing, and fixing already-identified bugs to finding and fixing intentional errors. The model responds to an increasing body of research in computing education about students' struggle in introductory programming courses to form useful conceptions of fundamental concepts (Sleeman et al. 1988; du Boulay 1989; Ma 2007; Sorva 2008; Kaczmarczyk et al. 2010).[1] Common to their struggles are activities dominated by "try-it-and-see-what-happens" or endless debugging on which learners cannot build their own understandings of what failed and why.

« 4 » The implementation of the proposed model had exercises that were designed iteratively, based on observations of students' interaction with the practice problems. The author "was assured that all students who attended the lab interacted with the key course concepts" (§37). Does this mean that those interactions scaffolded the construction of the powerful idea of how a program works and gets executed? Visualizations with codelens and tracing code by hand are activities that the model supports and seem quite adequate to help learners make concrete the abstract concepts of vari-

1| See also the paper "Parameter passing: The conceptions novices construct" presented by Sandra Madison and James Gifford at the Annual Meeting of the American Educational Research Association in Chicago, 24–28 March 1997. https://files.eric.ed.gov/fulltext/ED406211.pdf

ables and memory data storage, execution flow, and control structures. To what extent did learners' experiential activities help with building cognitive structures related to the notional machine? (Q1) I hypothesize that having built such structures, students might become more effective programmers, be more efficient in finding and fixing bugs, and write more sophisticated code.

« 5 » In pursuit of building the model's complementarity with constructionist teaching and learning, I find it important to highlight the learning context in which de-construction materials of worked examples and well-formed practice problems afford learners interactive activities of explaining, labeling, discerning (in various forms), fixing errors, and producing full solutions. The tangible objects that mediate these activities are of fine granularity – coding statements, execution states, activation frames, error messages, and building blocks, which are far removed from materials that spark creativity and satisfy personal preferences. More simply, de-construction materials do not bring about or give form to microworlds. The question is: What in the model of de-construction has the potential to engage CS learners in tinkering with atomic units of computation (whether a line of code or variable value) and motivate them to care about what they discover (missing statement, correct order of statements, or bug fix)? (Q2)

« 6 » The author's future plans that might investigate this question include the addition of "appealing graphics and gamification elements" to the debugging and reading/tracing activities, and "[h]aving students collaborate on de-construction activities, e.g., with Pair Programming" (§41). I find the latter direction of future research indispensable to creating constructionist conditions for student learning. In her examination of the views of constructivism, constructionism, and social-constructivism, Edith Ackermann (2004) stresses the importance of inter-personal relations to learning from experience by pointing out Lev Vygotsky's emphasis on "how shared cultural artifacts are used to help mediate" self-directed learning (Ackermann 2004: 22). In the context of de-constructionist pair programming, what social interactions are possible and how are they integrated with individual practice activities? (Q3)

« 7 » Bringing constructionism to college-level education, including CS, is long overdue (Sacristán et al. 2018). Griffin's model of de-construction provides a familiar starting point or "low floor" for teachers who value and have experience with worked examples (Morrison, Margulieux & Guzdial 2015) and practice problems tailored to address known misconceptions (Kaczmarczyk et al. 2010). The model's complementarity with constructionism has the potential to guide in the discovery of the "high ceiling" of much needed applications of constructionism in CS higher education.

References

Ackermann E. (2004) Constructing knowledge and transforming the world. In: Tokoro M. & Steels L. (eds.) A learning zone of one's own: Sharing representations and flow in collaborative learning environments. IOS Press, Amsterdam: 15–37. ▸ https://cepa.info/3894

Ben-Ari M. (1998) Constructivism in computer science education. In: SIGCSE Bulletin 30(1): 257–261.

du Boulay B. (1989) Some difficulties of learning to program. In: Soloway E. & Spohrer J. C. (eds.) Studying the novice programmer. Lawrence Erlbaum Associates: 283–299.

Kaczmarczyk L., Petrick E. R., East J. P. & Herman G. L. (2010) Identifying student misconceptions of programming. In: Proceedings or the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10). ACM, New York: 107–111.

Ma L. (2007) Investigating and improving novice programmers' mental models of programming concepts. Doctoral dissertation, Department of Computer & Information Sciences, University of Strathclyde.

Morrison B. B., Margulieux L. E. & Guzdial M. (2015) Subgoals, context, and worked examples in learning computing problem solving. In: Proceedings of the 11th International Conference on Computing Education Research (ICER '15). ACM, New York: 21–29.

Papert S. (1980) Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc., New York NY, USA.

Sacristán A. S., Baafi R. A. K, Sabin M. & Kamiskiene L. (2018) Constructionism in upper secondary and tertiary levels. In: Dagienė V. & Jasutė T. (eds.) Proceedings of the Constructionism 2018 conference. Vilnius University, Vilnius: 925–939.

Sleeman D., Putnam R. T., Baxter J. A. & Kuspa L. (1988) An introductory Pascal class: A case study of student errors. In: Mayer R. E. (ed.) Teaching and learning computer programming. Lawrence Erlbaum Associates, Mahwah NJ: 237–257.

Sorva J. (2008) The same but different – Students' understandings of primitive and object variables. In: Proceedings of the 8th Koli Calling International Conference on Computing Education Research (Koli Calling '08). ACM, New York: 5–15.

Sorva J. (2013) Notional machines and introductory programming education. ACM Transactions on Computing Education 13(2): 1–31.

Mihaela Sabin is a professor of computer science at the University of New Hampshire. Her current research includes computing education and curriculum development, with particular focus on broadening participation of students underrepresented in computing and expanding professional learning of computational practices for teachers. Sabin serves on the ACM Education Advisory Committee and chaired the ACM/IEEE Computer Society IT2017 task group, who authored the "Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology" report. Sabin has a PhD in Computer Science from the University of New Hampshire.

Received: 29 May 2019
Accepted: 31 May 2019

Deconstructionism – We Do Need More Success Stories

Pavel Boytchev
Sofia University, Bulgaria
boytchev/at/fmi.uni-sofia.bg

> Abstract • Deconstructionism in education is a powerful approach, which is still insufficiently researched. Griffin's presentation of the pedagogical aspect of deconstructionism is a success story, which may inspire others. Having experienced some of the issues, I also provide examples from my own practice.

« 1 » This commentary starts with a confession. Five years ago, I presented the concept of deconstructionism in education at the International Constructionism Conference in Vienna (Boytchev 2014). I was scared, as I was about to present a concept seemingly antagonistic to the main topic of the conference. The notion of deconstructionism had only been applied to philosophy and art (Boytchev 2015b: 367). Yet, deconstruction had been used in education for years, but has not been explicitly named and extensively researched. The scare quickly turned into happiness, as deconstructionism gained the attention of other researchers. Thus, this confession is my thank-you to Jean Griffin for providing examples of deconstructionism in the context of pedagogy.

« 2 » Griffin describes several approaches based on deconstructionism. Taking apart machines and appliances (§6), i.e., reverse-engineering, is one of the manifestations of deconstruction, as I have mentioned (Boytchev 2015a: 361). The mechanical dissection (§§8–10) is an accurate application of deconstructionism, although the word "dissection" is misleading. Dissection in medicine is a destructive process, as the topic of study or research is destroyed. Griffin, however, makes it clear that mechanical dissection includes [dis]assembling, re-assembling and developing (§§8-10). The Tod Phod Jod approach is very promising. It openly utilizes the deconstructionist approach. Griffin describes that Tod Phod Jod relies on "fun and intellectually stimulated experience" (§11). This leads me to my first question: What about considering deconstructionism as one of the inborn human behaviours, just like curiosity, creativity, etc? (Q1)

« 3 » Worked examples are another approach and I agree with the conclusion that learning is obstructed by cognitive overload (§13). Problems caused by excessive cognitive load or a cognitive barrier occur in the deconstruction phase (Boytchev 2015a: 362).

« 4 » In the context of programming code, comprehension must precede or occur at the same time as code writing (§16). This conclusion fits with how deconstruction and constructions are coupled. Understanding programming code is a deconstruction process, while writing code is construction. Interactive environments and the codelens component, described by Griffin in §25, provide support for faster and more active code comprehension.

« 5 » Programming topics are fruitful for learning from code and, especially, from intentional errors in the code (§§17f). Griffin provides sufficient references as to why this practice is controversial. My personal experience supports the positive impact of learning by mistakes – both intentional and unintentional mistakes. This approach is embedded in all my courses. One of them, called "Fundamentals of Computer Graphics" (FCG), has errors distributed in many of the lecture notes. This pulls students away from their default "write-only" mode – this is the tendency for students to take notes and to unconditionally soak up whatever information is presented during classes. In this respect I support Griffin's intention to collect more evidence about the potential benefits of learning from errors (§36).

« 6 » Let me recall my early years as a freshman. I enrolled in martial arts classes. The first few months we practiced only how to fall on the tatami in the least painful way, and how to get up immediately. Teaching martial arts has been fine-polished and optimized for many centuries; teaching programming has barely half a century of experience.[1] Can teaching programming learn from teaching martial arts? Do we need to teach students how to recover when their programs break down?

« 7 » I dream of a discipline that not only teaches the correct science, but utilizes the education power of historical misconceptions, failures and dead-ends. For example, in FCG there are historical sections about the catenary curve, studied in 1638 by Galileo, Jungius and Huygens (Kacmarynski 1931: 1). The curve is based on the hyperbolic cosine, but Galileo had mistakenly claimed that it was a parabola (Sonar 2018: 331). Another historical event is the origin of Bézier curves, developed by Paul de Casteljau, but named after Pierre Bézier (Vince 2006: 125) because of academic publishing problems.

« 8 » Griffin's experience in implementing deconstructionism as a pedagogy approach in learning a programming language is another interesting example of why and how deconstructionism is beneficial to education as much as constructionism is. However, I want to bring up a reminder about the importance of the careful designing of educational materials. Programming differs from Math at least in one vital aspect – most programming problems have many solutions – both completely distinct and brutally correct at the same time. This means that a "simple" task like that of locating the exact position of a bug may have multiple correct answers. Worked-out incorrect examples are a powerful arsenal, but they are risky for the teacher and for the students. This is also illustrated in Griffin's model for deconstruction, shown in Figure 4 in the target article and described in §26. All students' tasks within worked-out correct examples ("Explain or Label" and "Discern") could also be accessed via worked-out incorrect examples. The opposite is not correct. The "easier-harder" axis is important as it shows that problems with correct examples are easier than problems with incorrect examples even if they tackle the same student task.

« 9 » The design of suitable problems for the students is a difficult activity. It relies on *empathic deconstruction*. The teacher must foresee the deconstruction processes in students' minds in order to minimize potential failure during the deconstruction at lab-time.

« 10 » The research conducted by Griffin shows positive results. However, I would be more cautious about the conclusions extracted from the survey (§36). Such questions have an inherent risk of subjectivity. I would suggest the author follow up the aca-

1 | See Everett Murdock's lecture notes "History, the history of computers, and the history of computers in education," https://web.csulb.edu/~murdock/histofcs.html

Figure 1 • The Meiro environment.

demic development of these students and find out whether they have started using Python (of their own accord) in their other classes.

« 11 » Deconstructionism in education is heavily supported by technology. Griffin describes the advantages and disadvantages of using software like Runestone Interactive. I have similar experience with other software and 15 years ago I went along another path by deconstructing the educational software, and then reconstructing it by myself. This path is hard but it provides the freedom to create what you need and the confidence that you can do it when you need to. This is also related to Griffin's plan to introduce appealing graphics and gamification elements (§41). All my courses are "deluged" by my own graphics and interactive 3D models.

« 12 » Since 2018 the FCG course has been enriched by a gamified virtual environment called Meiro (Lekova & Boytchev 2018), which is used to present concepts during lectures and for self-education at home.

« 13 » Figure 1 shows thumbnails of a few hundred 3D models (left) and larger snapshots of the Meiro environment (right). The next version of the virtual environment will have gamified modules for (self-)evaluation and (self-)assessment (Boytchev & Boytcheva 2018). The gamification of Meiro is in the frame of the national project "ICT in Science, Education and Security."

« 14 » I agree with Griffin's conclusions that deconstructionism as a pedagogy is complementary to constructionism, and that the described experiment is a proof of the concept (§43). This commentary started with a confession, and ends with a request for a confession. Knowing a tool (any tool, either hardware, or software, or brainware) is complete, when you know when you *should not* use this tool. So, my second question is as follows: Have there been situations in which deconstructionism had a negative impact on the learning or on the learners? (Q2)

References

Boytchev P. (2014) Deconstructionism in education: A personal wandering towards constructionism. In: Proceedings of the third international constructionism conference. Austrian Computer Society, Vienna: 93–102.

Boytchev P. (2015a) Constructionism and deconstructionism. Constructivist Foundations 10(3): 355–363.
▸ https://constructivist.info/10/3/355

Boytchev P. (2015b) Author's response: Does understanding deconstruction require its deconstruction? Constructivist Foundations 10(3): 367–369.
▸ https://constructivist.info/10/3/367

Boytchev P. & Boytcheva S. (2018) Evaluation and assessment in TEL courses. In: Proceedings of the 44th international conference of applications of mathematics in engineering and economics. AIP Publishing, Melville NY: 020035. https://doi.org/10.1063/1.5082053

Kacmarynski J. (1931) The catenary. MS thesis. State University of Iowa. https://doi.org/10.17077/etd.o2vdr7xz

Lekova M. & Boytchev P. (2018) Virtual learning environment for computer graphics university course. In: Proceedings of the 12th international technology, education and development conference. IATED Academy, Valencia: 3301–3309.

Sonar T. (2018) The history of the priority dispute between Newton and Leibniz: Mathematics in history and culture. Birkhäuser, Basel.

Vince J. (2006) Mathematics for computer graphics. Second edition. Springer, London.

Pavel Boytchev is an associate professor and researcher at the Faculty of Mathematics and Informatics, Sofia University. His research interests are in the areas of developing university-level courses, educational software, computer graphics and multimedia. He has created numerous educational applications based on his own educational programming languages. He is an author of a dozen courses, hundreds of computer-generated video clips and thousands of computer demo programs in his areas of interest.

Received: 27 May 2019
Accepted: 31 May 2019

Constructionist Space Should Be Inclusive, Non-Dogmatic and Open to Multiple Sources

James E. Clayson
American University of Paris, France
james/at/clayson.org

> Abstract • Reading Griffin's clearly written article has led me to think about some questions that I would like to address particularly to members of the constructionist community. Constructionists have never sufficiently outlined a proper theory of learning; they have failed to document and report on their successes and failures in the classroom, thus making evaluation of their project difficult, and they sometimes seem unwilling to learn across disciplinary boundaries.

Theory, theorists and narrative

« 1 » Jean Griffin's article follows an earlier article by Pavel Boytchev (2015) that also addressed the notion of constructionism versus deconstructionism. Her opening argument is that physical manipulations might be usefully characterized as *either* putting things together (construction) or taking things apart (deconstruction). She then goes on to infer that this strict dichotomy observed in external events must also occur inside the body and mind.

« 2 » I question whether the author has perhaps made a "category error" by moving characteristics from one space to another without offering supporting evidence. For constructionists any manipulation of objects – whether in a constructive or deconstructive sense – can facilitate the construction of new knowledge. There is no dichotomy, and therefore Griffin's new theory of learning, called deconstructionism, would seem redundant.

« 3 » However, Griffin is right in opening up a discussion of learning theory because we constructionists tend to neglect it in favor of practice. We constructionists should be asking ourselves: why are we so allergic to theory when it can help us test and improve educational approaches?

« 4 » If we did spend more time discussing theory, we might see the irony in continuing to privilege what comes from an elite group of professional researchers over the narratives of students about their personal learning experiences.

« 5 » Seymour Papert (1991) came down clearly on the side of the student as theorist. He stated that when an individual builds and uses external artifacts to explore problems or issues, this activity can facilitate internal learning. It must be done, he said, in a public space and made sharable with others. As a teacher, I too have always assumed that to make artifacts understandable to others meant that they must first be understandable to the builder (Clayson 2018). Thus, the student builder becomes the principal actor in their own construction of knowledge and theory. The question of who should be encouraged to build theory is one of the most important unanswered questions in the constructionist community. My feeling is that we need everybody to be involved. We are all learners; we are all theorizers.

« 6 » Thinking more actively about who should build learning theories should also make us reconsider what kinds of learning we should be theorizing about.

« 7 » Most learning research takes place in schools. But what about other kinds of learning that might also be studied? Thomas Kuhn (1970), the historian and philosopher of science, introduced us to the importance and necessity of studying the critical times when old paradigms break down, new paradigms emerge, and scientists move on in new directions with renewed vigor and impetus.

« 8 » This kind of paradigm shift also happens to individuals at the personal level. For example, consider Seymour Papert's "personal gear story" (Papert 1982). Papert's love and obsession with the tiny gears of a toy car given to him by his father was a transformative event for the boy, which led him to explore and later "discover" many of the big constructionist ideas, such as simulation, body synchronicity, debugging and restructuration. Why haven't constructionists been able to integrate Papert's and others' personal learning stories into their overall theory of learning?

« 9 » Is it because we feel we must always move forward with "working to scale" and therefore tend to avoid idiographic study? For constructionists to gain a better understanding of transformative events, and to revalue their significance to individuals, we need to listen to what psychologists and sociologists can tell us (Turkle 2007, 2011; Bollas 1987). Their literature is rich with research on how narrative psychology, transformative events and objects, and the role of personal constructs can influence learning.

Constructionist/instructionist: False dichotomy

« 10 » Griffin's second argument is that constructionist courses, because they rely solely on student-directed exploration where learning – if any – is serendipitous, cannot be used for courses, especially in science and math, that address normative lists of skill and idea outcomes. She seems to imply that only instructionist courses can effectively teach to measurable outcomes.

« 11 » I do not feel that this is a proper reading of constructionist ideas and literature. For example, we have this clear statement from Wallace Feurzeig, one the pioneers of constructionism:

> "Constructionists want to build a critical mass of citizens who reject false and misleading educational dichotomies, who support instead the creation of learning environments that integrate the constructive ideas on both sides of tradition and reform, structure and freedom, knowledge and creativity, instruction and construction." (Feurzeig 2010: 18)

It is evident that constructionist teachers – like all good teachers – are expected to be non-dogmatic and to blend instructionist and constructionist tools as appropriate.

« 12 » The problem as I see it, and highlighted by the questions raised in Griffin's target article, is that we constructionist teachers have never fully documented and broadcast our mixtures of instruction and construction in high-school and university-level courses with clearly stated outcomes.

Problem sets and constructionist textbooks

« 13 » Griffin ends her article by giving an example of a computer science course that requires students to mix their building of new Python programs with close analysis of successful and less successful code snippets written by others (§§22–41). She labels this analysis of others' programs as

"deconstructionist." She compares, using very simple student satisfaction measures, the deconstructionist-augmented course with those traditional Python courses that lack such activity. She finishes the article by claiming that the positive results from her one experiment are a kind of "proof of concept" for her constructionist versus deconstructionist notions (§42).

« 14 » It seems to me that this is pretty standard fare for good pedagogy. It puts learning of new tools into contexts that are meaningful to both students and teachers. But what is missing from this account is the teacher's role as a participant in all collective learning experiences. For constructionists, learning in a public space requires everyone, including the teacher, to solve assigned problems, show how they did it, justify their approach, and be willing to engage in a general critique.

« 15 » Once again, responsibility lies with constructionists themselves. Why have we not produced teaching materials that privilege this collective endeavor, in which the teacher is an equal and active player in knowledge building? By relying on teaching materials that, by default, suggest that there is only one good approach, we are self-limiting our capacity for creativity and innovation.

Deconstruction: Where is Derrida?

« 16 » The final question prompted by the target article deals with the word "deconstructionism." Like the proverbial elephant is the room, it makes no sense to talk about deconstructionism without at least mentioning the contemporary philosopher, Jacques Derrida, whose life's work was defining deconstruction. Students in the humanities and social sciences are familiar with Derrida, whose use of the word "deconstruction" has entered world languages. However, I fear that many constructionists are not.

« 17 » Derrida (1985) proposed the radical notion that all texts are subject to unlimited interpretations and, hence, are subject to unresolvable ambiguities. Therefore, he seems to be suggesting that we can choose whatever reading is most useful to us, while acknowledging that this is only one possible choice among many. A similar view was espoused by George Kelly, the constructivist psychologist who developed the notion of "constructive alternativism" (Kelly 1955).

« 18 » To overlook Derrida's immense contribution to the notion of deconstruction is emblematic of our academic isolation from other disciplines. For constructionists to focus almost exclusively on science and math, without recognizing that social sciences, humanities and the arts also contribute to our mission is a major loss. How on earth can we talk about and build theories of learning without being informed by developments in other disciplines?

References

Bollas C. (1987) The shadow of the object: Psychoanalysis of the unthought known. Free Association Books, London.

Boytchev P. (2015) Constructionism and deconstructionism. Constructivist Foundations 10(3). ▸ https://constructivist.info/10/3/355

Clayson J. (2018) Artifacts, visual modeling and constructionism: To look more closely, to watch what happens. Problemos 2018: 8–23. http://www.journals.vu.lt/problemos/article/view/12345/

Derrida J. (1985) Letter to a Japanese friend. In: Wood D. & Bernasconi R. (eds.) Derrida and difference. Parousia Press, Warwick: 1–5.

Feurzeig W. (2010) Demystifying constructionism (Abstract). In: Clayson J. & Kalaš I. (eds.) Proceedings of Constructionism 2010. American University of Paris, Paris: 18.

Kelly G. (1955) The psychology of personal constructs. Volumes I and II. Norton, New York.

Kuhn T. (1970) The structure of scientific revolutions. Second edition. University of Chicago Press, Chicago.

Papert S. (1982) Mindstorms: Children, computers and powerful ideas. Perseus Books, Jackson.

Papert S. (1991) Situating constructionism. In: Harel I. & Papert S. (eds.) Constructionism. Ablex, Norwood NJ: 1–11. http://www.papert.org/articles/SituatingConstructionism.html

Turkle S. (ed.) (2007) Evocative objects: Things we think with. MIT Press, Cambridge.

Turkle S. (2011) Falling for science: Objects in mind. MIT Press, London.

James E. Clayson is Professor Emeritus at the American University of Paris (AUP), where he has taught applied mathematics and visual thinking for thirty years. He specializes in building computational environments where liberal arts undergraduates can explore the power of building personal models that link the visual, the qualitative and the quantitative. Jim was educated at MIT, the University of Chicago and the School of Oriental and African Studies (University of London).

Received: 23 May 2019
Accepted: 31 May 2019

Author's Response

De-Constructionism: Practice, Examples, Bugs

Jean M. Griffin
Temple University, USA
jeaniacgriffin/at/gmail.com

> Abstract • This response clarifies what de-constructionism is and explains how it differs from, complements, and to some extent overlaps with constructionism. It clarifies the distinction between the terms de-construction and deconstruction, and discusses student motivation and social interaction.

« 1 » Seymour Papert's protégé Idit Harel describes constructionism as follows:

> "Seymour coined the term to advance a new theory of learning, claiming that children learn best when they
> 1. use tech-empowered learning tools and computational environments,
> 2. take active roles of designers and builders; and
> 3. do it in a social setting, with helpful mentors and coaches, or over networks."[1]

« 2 » De-constructionism complements constructionism and is applicable to students of all ages. It has three guiding principles: provide ample, effective practice; guide students to deconstruct well-built examples in ways that promote metacognition; and develop negative knowledge by carefully crafting intentional bugs.

« 3 » Motivation is an important dimension of learning. To answer **Mihaela Sabin**'s Q2, the model for de-construction focuses on cognition rather than motivation. One way to address motivation is to pair de-constructionist activities with constructionist ones. Another way is to give students interesting things to deconstruct – well-built things that students are excited to learn about. These can be big things or physical things; they need not be limited to code segments as used in this study. Yet another way to foster motivation is to gamify the experience with rewards such as points, levels, or badges.

« 4 » Social interaction is another important dimension of learning that is critical to constructionism. Again, one way to address social interaction (Sabin Q3) is to pair de-constructionist activities with constructionist ones. This can also be accomplished through competitions or with a variety of collaborative approaches used in CS education. For example, with Pair Programming, students take turns using the computer keyboard. With Peer Instruction, the teacher poses a multiple-choice question. Students vote for an answer, discuss the question with their classmates, and vote again. Then the teacher reveals the correct answer. When designing Pair Programming or Peer Instruction activities, instructional designers can use the model for de-construction as a guide. POGIL (Process Oriented Guided Inquiry Learning) uses structured small-group learning where each student is assigned a role (e.g., manager, recorder, reporter). The group is guided to explore (deconstruct) a model via a series of questions, and then apply what they have learned by completing challenges. A contribution that de-construction makes to supplement POGIL is its emphasis on practice.

« 5 » I agree with the hypothesis that the de-constructionist approach can promote learning (**Sabin** Q1). Typically, learning is measured with pre/post assessments, where learning gains are calculated as the difference between pre-test and post-test scores (individually for each student or in aggregate for a collection of students). The challenge with evaluating de-construction is that it has so many components. Not only does it promote a variety of techniques (e.g., explaining, labeling, comparing, completing), it also promotes variation in practice (concrete-to-abstract, distributed, interleaved). This makes it challenging to design a controlled study. If there are differences in learning gains between a de-construction group and a control group, to which of these features can the differences be attributed? There is also the longitudinal factor – designs learned through de-construction may only prove useful much later during a design process. Because of these complexities I conducted an experiment to evaluate just one aspect of de-construction – learning from intentional bugs. Within the study described in this target article, I conducted an experiment where half the students got some practice problems with bugs while the other half got similar problems without bugs. Learning gains were measured with a pre/post placement test and exams. This experiment is discussed in a forthcoming article (Griffin 2019) based on my dissertation (Griffin 2018), accompanied by a collection of design principles to supplement the model for de-construction presented in this target article.

« 6 » To answer **Pavel Boytchev**'s Q2 about whether there have been situations in which de-constructionism has had a negative impact, I will share a personal experience. Several years ago, my team developed practice problems called *Debug'ems*™ that challenged students to find and fix bugs. When we gave them to advanced high-school students in a five-week summer program, the students enjoyed the challenge, especially because they involved animations with dramatic themes and the bugs had humorous consequences. When we gave the same problems to another group of high-school students – economically disadvantaged ones participating in short workshops – these students found the problems to be too difficult. Some would say that this was due to cognitive overload. I became aware that in this context, asking students to find and fix bugs was too difficult. It had the negative consequence of discouraging the students unnecessarily. To address this concern, I designed alternative activities similar to treasure hunts that provided more scaffolding. I created fun animations with code that the students were not yet capable of writing on their own. Students were guided to explore the code and answer challenging questions about it. These *Explore'ems* provided more scaffolding than the Debug'ems™ and the students enjoyed them (Griffin, Kaplan & Burke 2012). In the model for de-construction, Explore'ems would fall on the easier, left-hand side of the model, with the understanding that it is certainly possible to create some Debug'ems™ that are easier to solve than some Explore'ems.

« 7 » It is worth discussing the differences between de-construction and deconstruction. As to whether the impulse to deconstruct – take things apart – is an in-

1| "A glimpse into the playful world of Seymour Papert" by Idit Harel, EdSurge, 2016. https://www.edsurge.com/news/2016-08-03-a-glimpse-into-the-playful-world-of-seymour-papert

born human behavior like curiosity or creativity (Boytchev Q1), I am inclined to think so, but I will leave that for developmental psychologists to debate. I appreciate that James Clayson (§§16–18) mentions Jacques Derrida. Although I refer to Derrida in my dissertation (Griffin 2018) I did not do so in my target article. Derrida's philosophy of literary deconstruction is well established. This makes it problematic to use the term deconstructionism in another sense, as a pedagogy complementary to constructionism, as I initially did in 2012 (Griffin, Kaplan & Burke 2012: 4). I now use the term de-construction to clarify the distinction.

« 8 » Although de-constructionism's learning-by-taking-apart approach stands in contrast to constructionism, it makes explicit some of Papert's ideas and values that are frequently overlooked. For example, Papert valued learning by taking things apart. He recounted transformative childhood experiences of taking apart gears (Papert 1980). De-constructionism explicitly highlights learning-by-taking-apart by promoting specific techniques informed by cognitive psychology. These include explaining, labeling, comparing, discerning, and completing. Papert also viewed debugging as a powerful idea (ibid). De-constructionism explicitly promotes learning from bugs and the acquisition of negative knowledge, not just by learning from one's own mistakes, but by interacting with carefully designed intentional bugs. It also emphasizes effective practice, which is not addressed by constructionism. Practice with taking things apart and negative knowledge are critical in learning environments intended to develop technical expertise. Thus, although de-constructionism overlaps with constructionism to some extent, it is not redundant (Clayson §2), and there is not a strict dichotomy between the two paradigms (Clayson §1). Although constructionism was designed for young children, some argue that bringing it to older students is warranted (Laurillard 2002; Sacristán et al. 2018). Combining constructionism with de-constructionism is an effective way to bring constructionism to older students.

« 9 » Clayson's remarks (§10) about instructionism indicate a misunderstanding. I am a big fan of constructionism, especially in science and math courses. In settings where it is important for students to develop technical expertise, however, pairing it with de-construction will achieve the goals of topic coverage and skill development better than constructionism alone. To clarify, characterizing de-constructionism as opposite to constructionism (e.g., in the title of the target article) refers to the opposing approaches of learning-by-making and learning-by-taking-apart. This is a commonsense distinction that is accessible to teachers. De-constructionism is not meant to champion what Papert disdained about the ways in which math is typically taught in schools, devoid of personal utility, or *thingness*, or interest – all aspects of instructionism (Papert 1993, 1996). De-constructionism is inspired by hands-on activities with concrete things – by reverse engineering, mechanical dissection, and Tod Phod Jod. The main idea is to give students exposure to well-built examples that are relevant to them, which they can assimilate as design patterns and use later for their own constructions.

Acknowledgments

I thank the reviewers and editors for their thoughtful comments as well as my dissertation committee members, Catherine Schifter, Kristie Newton, John Dougherty, and Julie Booth.

References

Griffin J. M. (2018) Learning to program with interactive example code (with and without intentional bugs). Doctoral dissertation, Temple University, USA.

Griffin J. M. (2019) Designing intentional bugs for learning. In: Proceedings of the first UK and Ireland Computing Education Research conference. ACM, Canterbury UK, in press.

Griffin J. M., Kaplan E. & Burke Q. (2012) Debug'ems and other Deconstruction Kits for STEM learning. In: Proceedings of the second IEEE integrated STEM education conference (ISEC '12). IEEE Press, Piscataway NJ: 1–4.

Laurillard D. (2002) Rethinking university teaching: A conversational framework for the effective use of learning technologies. Second edition. Routledge, London.

Papert S. (1980) Mindstorms: Children, computers, and powerful ideas. Basic Books, New York.

Papert S. (1993) The children's machine: Rethinking school in the age of the computer. Basic Books, New York.

Papert S. (1996) An exploration in the space of mathematics educations. International Journal of Computers for Mathematical Learning 1(1): 95–123.

Sacristán A. I., Kaminskiene L., Sabin M. & Baafi R. A. K. (2018) Constructionism in upper secondary and tertiary levels. In: Dagienė V. & Jasutė E. (eds.) Constructionism 2018. Vilnius University, Vilnius: 932–946.

Received: 30 June 2019
Accepted: 10 July 2019