BLOG @CACM

The *Communications* Web site, <http://cacm.acm.org>, features more than a dozen bloggers in the BLOG@CACM community. In each issue of *Communications*, we'll publish selected posts or excerpts.

Follow us on Twitter at <http://twitter.com/blogCACM>

DOI:10.1145/3077227

<http://cacm.acm.org/blogs/blog-cacm>

Balancing Teaching CS Efficiently with Motivating Students

Mark Guzdial suggests a new balance is needed in computer science education between discovery learning and direct instruction.

Mark Guzdial
'Figure It Out' Isn't Enough: Striking a New Balance Between Understanding, Problem-Solving, and Design in Introductory CS Classes

<http://bit.ly/2jvLDdO>
January 9, 2017

A computing educator has to balance teaching efficiently and motivating the student. Efficient teaching means teaching abstractly, emphasizing practice, and preferring direct instruction over having students “figure it out.” Motivating the student means giving the students authentic situations, real-world complexity, and reasons to practice.

I recently wrote an essay describing this tension (<http://bit.ly/2nFRuGZ>). Herbert Simon (one of the three authors of the *Science* article first answering the question “What is Computer Science?”; <http://bit.ly/2nFIzpf>) strongly believed in direct instruction, rather than problem-solving. Having students “figure out” the solution for themselves, to discover solutions to problems, was an in-

efficient way to learn for Simon, and his concern was for motivation:

When, for whatever reason, students cannot construct the knowledge for themselves, they need some instruction. The argument that knowledge must be constructed is very similar to the earlier arguments that discovery learning is superior to direct instruction. In point of fact, there is very little positive evidence for discovery learning and it is often inferior ... Because most of the learning in discovery learning only takes place after the construct has been found, when the search is lengthy or unsuccessful, motivation commonly flags.

A teacher of introductory computer science faces the tension between letting students figure out complex situations and telling students the answer. I'm going to describe the tension using a generalized, perhaps even stereotypical description of what students want. As Valerie Barr has pointed out (<http://bit.ly/2nFRSFx>), teachers need to understand the students who are in their classes, not generalizations. While the generalization I'm using doesn't accurately describe all students, the gen-

eralization matches how CS teachers think about their students (<http://bit.ly/2nYA3nE>), which does explain what we do in our classes.

A student who takes an introductory computer science course wants to *make* something. Even if the student doesn't want to become a professional software developer, they want to *create* software, to *design* something digital. We want to go from where they are to producing something interesting. The challenge (as Briana Morrison and I describe in the November 2016 *Communications*, <http://bit.ly/2iIFeEc>) is that students enter CS class with less background in the discipline than any other STEM field. It's difficult to design when you don't understand the medium that you're designing with.

We know students need to develop an understanding of what the computer does when it executes programs. Computing education researchers call that the *notional machine* (see the report from a Dagstuhl Seminar group on student learning about notional machines at <http://bit.ly/2oy1KoO>). To design and

debug programs, students need to develop a mental model of the notional machine. We have a poor track record in helping students be able to trace and predict program execution, as Raymond Lister has been exploring in his research (see <http://bit.ly/2n7A8Xr>).

Maybe our students don't develop enough understanding of the notional machine because our introductory courses don't make program understanding a key learning goal. Students *need* to understand, but they *want* to build. We can teach for understanding, but that's harder to do in authentic, complex learning situations, as Simon and others have pointed out (<http://bit.ly/2nFRuGZ>).

Perhaps in response, many introductory computer sciences take a middle ground and focus on *problem-solving* (see examples at <http://bit.ly/2oVyXXu> and <http://bit.ly/2n7yu80>). Some teachers even define computer science as “algorithmic problem-solving” (<http://bit.ly/2nYJhQP>, which doesn't appear at all in the Newell, Perlis, and Simon definition of the field at <http://bit.ly/2nFIzpf>). Teachers can get students to realize they have to solve problems to create software, so they emphasize how to solve problems with programs and algorithms.

A focus on problem-solving is a rational way to strike a balance between getting students to understand programs and their desire to build. We give students problem statements (describing things to build), and we teach them how to go from the problem statements to a working program. We teach them how to design with objects, and how to analyze problems for the data structures within them.

The problem is that teaching problem-solving is not the same thing as teaching for understanding, and empirical evidence suggests it isn't working. John Sweller showed years ago that more problem-solving doesn't lead to greater understanding. Problem-solving creates enormous cognitive load that interferes with learning to understand (<http://bit.ly/2nAufgV>). If we want students to understand more, we have to teach for understanding. In my book *Learner-Centered Design of Computing Education* (<http://bit.ly/1JYLeUz>), I describe some of the evidence that students are not developing an under-

We need to create learning situations where we ask students to practice program reading, to predict program execution, and to understand program idioms.

standing of programs and developing a mental model of the notional machine.

To teach for understanding, we would give students worked examples and ask them questions about the examples, ask students to predict outcomes or next steps in a visualization (<http://bit.ly/2nAs5xN>), or ask students to solve Parson's Problems. **We would do far less of giving students a problem they've never seen before, and asking them to generate a brand-new program to solve that problem.**

At ICER 2016, Briana Morrison, Lauren Margulieux, and Adrienne Decker presented a replication study showing that introductory students miss important details in problem statements, but they figure them out when the students reach their second CS course (<http://bit.ly/2n7BOjH>). Morrison thinks it takes students that long to develop their understanding so they are more effective at problem-solving. We could perhaps achieve better understanding earlier, but we'd have to teach for understanding. We computer science teachers tend to underemphasize program comprehension, because it's boring for us—and it's easy for computer science teachers. It falls in our expert blind spot. A focus on understanding can be boring for the students, too, because it's not about *making* stuff.

We need a new balance point. We need to do more to get students to understand. They need to build, too, because that's important for student motivation. We need to create learning situations where we ask students to practice program reading, to predict

program execution, and to understand program idioms. More problem-solving might need to wait until student understanding catches up.

Comments

I have to question the assumptions that teaching for understanding is the right goal. There are many aspects in the applications I build that I don't understand. But I don't need to.

“Understanding” is such a slippery goal, whether we talking about understanding computers or Shakespeare. It only becomes well defined when we talk about when you need it, for example, to find a bug, or adapt a solution to a new problem. But then we're back to skills in designing and problem solving as the primary goal.

I do love the notional machine concept though, at least as I “understand” it so far. Which means how I see applying it is thus: to help students when they are struggling in debugging or design, help them articulate and develop a better, partial, locally useful, notional machine.

—Christopher Riesbeck

Hi Chris,

I bet that we would agree that the depth of the understanding is the question. I use my computer all the time without thinking about transistors.

Think about it in terms of Bloom's taxonomy. The lowest level of learning is simply being able to repeat what was heard. Later levels include being able recognize the right thing and to predict. The highest levels are synthesis and problem-solving. Surely those lower levels matter in computer science. We can't ONLY expect students to perform at the highest levels. We also have to teach for those lower levels, too.

—Mark Guzdial

Dear Mr. Guzdial,

Many thanks for your article on CS for all. Being retired, I spare my time with a group “fighting against school dropping out.” We receive students after their school hours and help them doing their homework. It works fine! I'm planning, may be some times from now, to introduce them to CS. Your article inspired me. Again, many thanks!

—Mario Beland

Mark Guzdial is Director of Contextualized Support for Learning at the Georgia Tech College of Computing.

© 2017 ACM 0001-0782/17/06 \$15.00