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.

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

DOI:10.1145/3344262 http://cacm.acm.org/blogs/blog-cacm

Why Programmers Should Curb Their Enthusiasm, and Thinking about Computational Thinking

Yegor Bugayenko ponders the dangers of "hazardous enthusiasm," while Mark Guzdial considers whether the need to teach computational thinking can be "designed away."

Yegor Bugayenko Hazardous Enthusiasm and How Eagerness Can Kill A Project

http://bit.ly/2LHzuLq
June 27, 2019

Programmers are constantly contributing to my open source projects (all of my projects are open source, FYI). Some are volunteering their time, others are paid through Zerocracy. While I have worked with a lot of great developers over the years, I have also come across a number of people afflicted with what I call "hazardous enthusiasm."

These people have energy and often the skills, but are overzealous and don't know how to break down their changes and deliver them incrementally. People afflicted with hazardous enthusiasm frequently want to tear down and rebuild the entire architecture or implement some other huge changes, often simply for the sake of doing so. Let's take a closer look at what I mean.

Let's say a new developer joins the team. At first, he checks all the boxes. He knows his code, he's got energy, he's easy to communicate with, and he's putting in time, submitting new tickets and offering useful suggestions. During those first few days, he seems like a gift from the heavens.

As he learns more about the project, the hazardous enthusiasm starts to creep in. Instead of tickets with helpful suggestions, he hits me up on Telegram with a bold claim: the architecture is a complete and utter mess, and I've got just a matter of weeks before the project will implode.

I counter with a polite reassurance that I understand, but before even hearing me out, he's already suggesting that we re-do everything from scratch. At the very least, he suggests we trash a collection of objects, and replace them with a singleton and a particular ORM library. Of course, he's been using these for months, and they're amazing and as soon as I see everything in action I'm going to be floored and, and, and ...

Now at this stage, there's a lot I will probably want to say. I could remind him that I am an architect myself, and that I have a long string of successes under my belt. I might point out that we've been working on this project for some time and that so far development is progressing at a comfortable pace.

Often, however, I say very little and instead ask him to submit a ticket. I offer an assurance: I'll review his suggestions as soon as possible. And I casually remind him that I am an architect, and in fact the architect for this project. In an ideal world, he'd accept that and follow up some incremental changes. More often, he claims that he'll show me how it's supposed to be done.

A few days later, he hits me up with a huge pull request. There are tons of changes, and some of them actually look quite interesting. The problem is, a lot of the suggestions are all but antithetical to the principles I've embedded into the existing architecture. I know he's put a lot of time into his project, but I have to reject the pull request anyway.

Can you guess what happens next? The developer, once a godsend, simply ups and disappears. You see, I'm

the bad guy here. I am evil and anti-innovation and closed-minded. How dare I not scrap an entire project and start over!? I've been through all of the above time and time again.

The sad thing is, that developer probably could have made a lot of useful contributions. Sometimes we come across incompetent developers, but a lot of times they're actually great from the technical perspective; what they're lacking is an ability to microtask.

Developers jumping onto new projects need to know how to break down changes into small, digestible chunks and then deliver them incrementally. Instead of pushing out one huge chunk of changes or trying to completely upend the entire project, they need to set their sights lower. As an experienced and successful architect, I'm not going to allow someone to completely implode a project in their first week.

Maybe I'm evil. More likely, the developer has been struck with a case of fatal enthusiasm. Although they want to do the right thing, they are way too eager and overly zealous. Every fix has to be implemented in one pull request and there's no time to wait. Any incremental improvements simply won't be acceptable. Remember, in their view, time is running out and the project is only weeks from failing anyway.

So why don't I just step aside and let them fix the code the way they want? Maybe they're simply a better architect then me. But here's the thing: being a successful architect requires microtasking. As an architect, you have to manage changes, and you have to implement them gradually. This is a basic necessity in a dynamic, collaborative work environment.

The moment a developer comes to me and tries to upend the entire project just a few days in, I already know they are going to struggle with incremental change. That means they're going to struggle in the architect's seat, so I can't exactly hand over the keys to the whole venture.

So no, you are not being evil or closed-minded when you reject hazardous enthusiasm. You are being prudent, wise, or whatever you want to call it. Most importantly, you're being a good architect.

Mark Guzdial
A Design Perspective on Computational Thinking

http://bit.ly/2JkL3q2
June 9, 2019

Computational thinking was popularized in a March 2006 column in *Communications* by Jeannette Wing. In 2010, she published a more concise definition (see her article about the evolution of these definitions at http://bit.ly/2Xwr1Nr):

Computational thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent (Cuny, Snyder, and Wing, 2010).

I have been thinking a lot about this definition (see the BLOG@CACM from last September at http://bit.ly/2S437aS, and my April blog at http://bit.ly/2YBljuV). This is a definition most people can agree with. The problem is when you use it to define curriculum. What does it mean to represent a problem in a form that can be effectively solved by a computer? What do we teach to give students that ability?

Computers are *designed*. The problem form *changes*. We can make computers *easier* to use.

Human-computer interface designers and programming language designers are all about making it *easier* to represent problems in a computable form. A good user interface hides the complexity of computation. Building a spreadsheet is much easier than doing the same calculations by hand or writing a program.

I have been digging deeper into the literature on designing domain-specific programming languages. The empirical research is pretty strong. Domain-specific programming languages lead to greater accuracy and efficiency than use of general-purpose languages on the same tasks (as an example, see http://bit.ly/2NHhFPh). We are learning to make programming languages that are easy to learn and use. Sarah Chasins and colleagues created a language for a specific task (Web scraping) that users could learn and use *faster* than existing users of Selenium could solve the same task (see the blog post at http://bit.ly/2XPd9Sx).

So, what should we teach in a class on computational thinking, to enable students to represent problems in a form that the computer can use? What are the skills and knowledge they will *need*?

▸ Maybe iteration? Bootstrap: Algebra (http://bit.ly/2YMinvK) showed that students can learn to build video games *and* learn algebraic problem-solving, without ever having to code repetition into their programs.

▸ Booleans? Most students using Scratch don't use "and," "or," or "not" at all (see the paper at http://bit.ly/2L8ORwL). Millions of students solve problems on a computer that they find personally motivating, and they do not seem to need Booleans.

Our empirical evidence suggests even expert programmers really learn to program within a given domain. When expert programmers switch domains, they do no better than a novice (see the post at xhttp://bit.ly/2NEZidz). Expertise in programming is domain-specific. We can teach students to represent problems in a form the computer could solve in a single domain, but to teach them how to solve in multiple domains is a big-time investment. Our evidence suggests students graduating with a four-year undergraduate degree don't have that ability.

Solving problems with a computer requires skills and knowledge different from solving them without a computer. That's computational thinking. We will never make the computer completely disappear. The interface between humans and computers will always have a mismatch, and the human will likely have to adapt to the computer to cover that mismatch. But the gap is getting smaller all the time. In the end, maybe there's not really that much to teach under this definition of computational thinking. Maybe we can just design away the need for computational thinking.

Comments:

I wonder how well Khan Academy's approach to teaching computational thinking works, since it seems to be more interactive and can be connected to other skills (if there are courses for them): https://www.khanacademy.org/computing

—Rudolf Olah

Yegor Bugayenko is founder and CEO of software engineering and management platform Zerocracy. **Mark Guzdial** is a professor of electrical engineering and computer science, and engineering education research, in the College of Engineering, and professor of information in the School of Information of the University of Michigan.