SIGSOFT Influential Educator Award

Thoughts from a Not-So-Influential Educator

Greg Wilson
RStudio PBC
gvwilson@third-bit.com
DOI: 10.1145/3402127.3402136
https://doi.acm.org/10.1145/3402127.3402136

I was honored to receive ACM SIGSOFT's Influential Educator Award[1] for 2020 this past April. I was also surprised: while I think I've helped scientists through Software Carpentry and other projects, nothing I've done in the last twenty years seems to have had much influence on software engineering.

It isn't for lack of trying. In the early 2000s I began teaching classes at the University of Toronto. One was titled "Software Architecture", and after three very frustrating offerings I told the department they should cancel it. The problem was that the half-dozen textbooks I read with "software architecture" in their titles spent hundreds of pages explaining how to elicit architectural requirements and how to document architectures, but devoted less than 20 pages in total to describing actual systems. Students memorized what I put in front of them and passed their exams, but it had no impact on how they thought or what they built.

In frustration, I asked several dozen well-known programmers to describe the most beautiful pieces of software they'd ever seen. Their responses were published as *Beautiful Code*[2], which won the 2008 Jolt Award for best general book. I found every chapter fascinating, but the sheer diversity of examples they chose meant it still wasn't what I needed for teaching.

Amy Brown and I therefore organized a follow-on series called *The Architecture of Open Source Applications*[3]. In the first two volumes, the creators of fifty open source projects explained how those systems were structured and why. The third volume looked at performance issues, while the fourth presented small tutorial re-implementations of spreadsheets, web servers, and other applications. Ironically, by the time I had what I'd wanted to teach I had left academia behind.

In parallel with these efforts Andy Oram and I edited another book called *Making Software*[4], in which empirical software engineering researchers described their favorite discoveries and explained why we believe they're true. Jorge Aranda and I followed that up with a website called *It Will Never Work In Theory*[5] where we presented brief summaries of empirical research that we hoped practitioners would find interesting. The site was active for six years, but never reached critical mass: most researchers weren't interested in explaining their work to laypeople, and most working programmers didn't find their work relevant.

I was (and am) proud of these efforts, but as I said at the outset, they have had no significant impact on software engineering education. We still don't give undergraduates guided tours of real applications to acquaint them with prior art and give them models to draw on when it's their turn to make design choices. Similarly, our students finish their degrees knowing little about key results from empirical software engineering research and even less about the methods used to obtain those insights. They are architects who have never seen an elevator shaft and biologists who don't know *why* we believe in evolution because that is who we have taught them to be.

So, what would I change in 2020? First, I believe we should teach students design by having them study the great works of the past, just as other disciplines do. They should spend at least a full course reading code and building working models of common applications in order to become familiar with prior art and prior thinking. In their assignments, they should compare and contrast undo/redo handling in Emacs and Vim or the data structures beneath Git and Mercurial, or create programs like Mary Rose Cook's *Gitlet*[6], Matt Brubeck's *layout engine*[7], or Conor Stack's *little database*[8].

Second, we should teach our students data science using software engineering examples so that they understand how science works and are familiar with what we already know. Are some programming languages actually easier to read than others? (Yes.) Does test-driven development actually produce better code faster? (Probably not.) Other engineering disciplines teach students to tackle questions like these by collecting and analyzing data, and we should too. Conferences like *Mining Software Repositories*[9] have made a lot of raw material available and produced some fascinating results for students to replicate (or refute). Such a course would be culturally defensible---no one ever got fired for saying that computer science students should learn more math---and from a self-interested point of view, graduates of such a course would be better prepared for grad school.

[1] http://www.sigsoft.org/awards/influentialEducatorAward.html

[2] http://shop.oreilly.com/product/9780596510046.do

[3] http://aosabook.org

[4] http://shop.oreilly.com/product/9780596808303.do

[5] http://neverworkintheory.org/

[6] http://gitlet.maryrosecook.com/

[7] https://limpet.net/mbrubeck/2014/08/08/toy-layout-engine-1.html

[8] https://cstack.github.io/db_tutorial/

[9] http://www.msrconf.org/

At the same time, there are several things we should stop doing, one of which is UML. Marian Petre's award-winning UML in Practice[10] analyzed why most developers don't use it; quoting one of her subjects, "UML is to the modeling we do every day as Latin is to the language we use every day." Mike Hoye likens it to trepanation: medical students should know why it exists, why we thought it would work, and why we now do something else, but that's an hour, not a semester. If we are going to teach modeling, we should teach something that is both rigorous and useful: Daniel Jackson's *Software Abstractions*[11] or Hillel Wayne's *Practical TLA+*[12] would be excellent choices.

I would also shelve team project courses that require students to pretend to follow a name-brand process like Scrum. I say "pretend" because faculty rarely coordinate homework deadlines. As a result, students can't work in the steady, focused fashion that such processes assume (and that we crave for ourselves). What simulated project courses actually teach is that crunch mode is normal, and the most useful skill most students take away from them is how to dissemble. If we want students to be able to work in teams then we should teach them how to run meetings, manage conflict, and create inclusive environments. Add a few lectures on liability, employment law, and intellectual property, and they will be better prepared for what comes next than most of us were.

Changing an established curriculum is hard, but we owe it to our students, and the two courses I've described above would be fun to create. If you'd like to help do so, please mail me at gvwilson@third-bit.com.

My thanks once again to SIGSOFT for this award; I hope you and yours are safe and well.

- *Greg Wilson is the co-founder of Software Carpentry, a member of the Education team at RStudio PBC, and the author of Teaching Tech Together (http://teachtogether.tech)*

[10] http://oro.open.ac.uk/35805/

[11] https://mitpress.mit.edu/books/software-abstractions-revised-edition

[12] https://www.apress.com/gp/book/9781484238288