Programmed to Succeed: Why Good Coders Should Read Great Literature

An impassioned (but still resolutely logical) plea to Stanford University, following the institution's decision to no longer provide lectures on literature to students destined for careers outside of the humanities.

Clementine Jacoby
Philosophy & Literature
Winter, 2015

If war is too important to leave to generals and the economy too important to leave to bankers, then coding is too important to leave to computer science departments. Yes, even Stanford's. The programmers of the next generation will be tasked with implementing the "internet of things." They will see the developing world come online; they will be tasked with automating cars, delivering goods by drone, and trying to detect cancer with wrist watches. The great programmers of the next generation will require some critical training the computer science department isn't equipped to offer. In fact, the literature department may be our best chance at preparing these students for the apparently intractable (or even unknown) problems that lie ahead.

To be sure, we already have a lot of sound defenses of reading that take a very general form. Toni Morrison defends the humanities in Song of Solomon with, "It's not about you living longer. It's about how you live and why." Joshua Landy says that great literature "enable[s] us to clarify ourselves to ourselves." W.E.B. Du Bois, in Souls of Black Folk, aims straight at universities: "The true college will ever have one goal – not to earn meat, but to know the end and aim of that life which meat nourishes."

Stanford CS undergrads are definitely earning plenty of meat these days. But how is the nourishment part coming along? It could be better, and great literature could be a big part of that improvement. Nevertheless, I can think of three reasons our army of undergraduate computer science majors might fail to heed the battle cry of "better books":

1. We engineers do not deal in vaguery. Your arguments for well roundedness, global citizenship, and elevated civil discourse are appealing, but not concrete enough to inspire change in our cool, analytical minds.
2. Your arguments seem important, but cannot compete with the urgency of problems that engineers are tasked with (see above).
3. The time invested in studying literature doesn't provide a sufficiently valuable return. I won't make money by studying literature, and nobody will pay me for having studied it.

I'd like to counter these postulated stances by arguing that there are reasons—concrete and urgent and even lucrative—that programmers should study literature. I claim that studying literature can help programmers do their job better than they otherwise would. The argument here differs from the classic stance that the humanities "leaven" the scientist—giving her a bit of elevation and humanism to complement her [insert stereotype]. Instead, I will argue that studying literature actually makes the scientist (or the programmer, in this case) better at what they do.

Proust for Programmers

Reading literature teaches us to embody the perspective of "the other", leading to better user-centered design and to software that delights.

When a person has become good at something, how do they think their way back into the mind of someone just encountering the subject? This is a core problem for writers

(and presumably literature faculty), but it's also a core problem for programmers and designers. There are many ways to define great software, but most convincing definitions include some notion of empathy. Here's my current favorite:

> "The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct."
>
> – Donald E. Knuth, Selected Papers on Computer Science

Software is user-driven. Deeply so. It's built to work, where work is defined by whomever or whatever consumes it. This consumer could be a human, a compiler, or a web browser. A compiler is actually another program that transforms code written by a programmer into something that the computer can interpret. Compilers place stringent rules on programs, dictating what will and will not work, just as the English language places strict constraints on our grammar and spelling. But these are low-level constraints. Sure, learning to spell is a requisite step on the path to writing the great American novel or becoming Poet Laureate. And so too all programmers must satisfy the compiler in order to do anything worthwhile. But to characterize what programmers do in terms of satisfying the demands of a compiler would be to woefully underspecify the task—it would be like saying that great writing requires proper spelling.

So how do we define great software? To merely say it's "any code that successfully compiles" sets the bar too low. Instead, great software makes the user feel like "whoever wrote this software had me in mind when they wrote it, and they must love me.[1]" This describes how I feel about the creator of Gmail: she knows my definition of a priority email; she directs spam to the trash bin without my help. These feel like acts of love or at least empathy. I'm aware that Gmail in fact has many authors, and that none of them know me, but this is how the software makes me feel, and that makes it great. Better yet, this definition approximates the ambitions of those who aspired to write it. Programmers are immodest. They see the applications of great software as unbounded in scope. Great programmers see every problem as within their purview. Anyone who has spoken to a programmer about politics knows that this is true.

With that definition of great software, let's ask how studying literature helps us write it. One answer is that literature lets us try on another worldview. Reading is an exercise in putting aside our own perspective in order to embody the perspective of the author or a character. We spend hundreds of pages seeing the world through new eyes. Empathy—in any setting—is about placing the importance of individual perspective above other aspects of external reality, however accurate those may be. To empathize we must prioritize the perspective of the other, which means we must first parse out what the individual's perspective is.

---

[1] Lloyd Tabb. *Great Software is an Act of Empathy.* August 19, 2013.

Proust thinks that literature, especially poetry, can help us do that. Poetry uses imagery to reveal the unique distortions that an individual's perspective imposes on the world[2]. Take an example from Proust's The Swann's Way. Marcel, his aspiring-poet character, describes the steeples of a church twice, calling attention to the distinction between the steeples considered objectively and the steeples as he, Marcel, sees them. The first is a straightforward recounting of the facts; the second is a poem. Seeing these two accounts side-by-side allows the reader to subtract the perceived from the objective, to hone in on exactly how and where Marcel's account colors our perception. Proust provides us with everything we need to unearth Marcel's perspective.

The side-by-side view helps us extract what Marcel feels from what he sees. We don't always get this from a writer, but poetry often provides enough information to get us started in getting outside our own angle of vision. As a reader, when we try to understand Proust's imagery or Baudelaire's metaphors, we do this. Or when we view the works of the impressionist painter, who the character Elstir tells us, "reproduce[s] things not as he knew them to be but according to the optical illusions of which our first sight of them is composed,[3]" we are also learning to reconstruct an other's perspective. By studying poetry, we learn to recreate our first encounter with an object, working backwards from what we know toward what we felt initially.

Programmers need to be able to do this. We need a kind of "synecdoche of programming", where our little subjective initial piece gives us a usable angle on the big, thorny, imposing objective whole. Like Proust, we also need to be able to set the technical reality to one side and the user experience on the other, construct a God's eye view of our creation. We need to be able to separate what we know to be true (because we built the thing) from what users feel when they encounter it. It doesn't matter if our new interpolation algorithm on Google Maps is objectively more accurate at deeper zooms. If it makes exploring the map feel sluggish, users will be outraged. Understanding there is a difference between what is actually there and what is perceived to be—and acknowledging the primacy of the latter—makes a good programmer great. In the software market, where the energy is high and the distribution barriers are low, building something people love using is a much shorter path to success than building something technically perfect.

Unfortunately, most people can't articulate the biases, values, and experiences that constitute their perspective. Fortunately, reading literature can teach us how to unearth them. In The Swann's Way, Marcel's metaphors reveal his point of view. His use of imagery shows us how he sees the world, hinting at his "private domains of association," which are based on his own "idiosyncratic passions and attachments"[4]. Joshua Landy groups scallops and cilantro together as items that repel the tongue (though not mine). Like Landy's, all minds organize experience not according to an objective set of transcendent categories, but according to a unique perspective,

---

[2] Joshua Landy. Philosophy as Fiction: Self, Deception, and Knowledge in Proust. Oxford and New York: Oxford University Press, 2004.
[3] Marcel Proust. Within a Budding Grove. 1919.
[4] Joshua Landy. *Philosophy as Fiction*.

dictated by our tastes and experiences. Associations matter. A web developer who neglects the differing associative properties of Papyrus and **Braggadocio** will cause users to revolt.

By contrast, programmers who understand their users can delight them. Understanding users means better design, and better design leads to software that meets a genuine user need rather than software that joins the ever-growing ranks of "technology awaiting a use case." In the face of coders' learned tendencies towards narcissism, it's already a big advance to write code like Marcel writes poems—that is, to at least remember there's a potential subjectivity gap. If we can write code like Proust write prose—that is, to move back and forth from their eyes to our own—we might stand some chance of meeting those big challenges noted at the outset. But this will require more than clarity of vision. It will also require techniques designed to ferret out our own weaknesses. Fortunately, the humanities have some lessons here, as well.

Debugging as Dialectic

Reading literature teaches us to perceive clearly, to spot holes in arguments, and to become impervious to rhetoric (even our own).

When I started debugging code (nanoseconds after I started writing code), I was struck by how similar the process was to fixing a bad argument. Argument reconstruction is a technique used by philosophy students to find flaws in arguments. It remains the closest thing to debugging that I've found outside of the practice itself. In both cases, we're faced with the unfortunate fact that the program (or the argument) would work if all of our assumptions were met, and yet it does not work, and so one of our assumptions must have been violated. Again, in both cases, it's often true that each individual step seems reasonable, and yet we've arrived at an absurd conclusion.

Like in an argument, and unlike in bridges, buildings, and other feats of engineering, the programmer often can't immediately see what's wrong in a buggy program. Usually, the fastest way to debug is to line up all of the assumptions, poke them one-by-one, and see which one falls over. In order to do so, you need to know:

1. What your assumptions are
2. How to poke them
3. What it looks like for one to fall over

Reading literature—specifically dialectics—trains us in all three pursuits.

Consider Gorgias. For Landy and others who see Gorgias as an example of Platonic Irony[5], the author lets Socrates make flawed arguments for the benefit of the reader.

[5] Landy, Joshua (2007). Philosophical Training Grounds: Socratic Sophistry and Platonic Perfection in "Symposium" and "Gorgias". *Arion* 15 (1): 63-122.

His cloudy thinking helps us see clearly. Plato allows Socrates to stumble so that the reader may triumph in recognizing the crumbling foundation. For Plato, there is a place for bad arguments in good literature. Spotting bad arguments in literature trains us to spot them faster in the wild. By letting Socrates succumb to obvious fallacies, Plato invites the audience to detect and correct them. The idea is that this sharpens readers' analytical skills, a bit like the lion teaching the cub how to hunt by letting it chase butterflies or chew on its siblings. Furthermore, Plato invites us to critique even those arguments born of highly reputed minds (like Socrates').

It's this ability that programmers use when debugging code. When programmers speak of "bugs," they're referring not to unfinished or broken code, but to violated assumptions. A bug is just a naturalistic metaphor for a violated assumption. For example, I have a penchant for unintentionally dividing by zero. Every fifth graders knows you can't do this, but I keep doing it anyway. So in calculating something like speed = miles/hours, I assume that my denominator is non-zero (because I am supposedly smarter than a fifth grader). When with depressing frequency I find I have put in a zero after all, the program breaks and my grades suffer. It turns out I am not alone. In 1999, a similarly silly assumption caused NASA to lose the Mars Orbiter. Lockheed's software produced output in pound-seconds units instead of the metric units of newton-seconds that NASA was expecting. The resultant trajectory brought the orbiter too close to Mars, and it disintegrated upon contact with the atmosphere. They would have been better off putting in a zero; at least the program would have broken and they could have found their mistake. When all of the programmer's assumptions are met, the program will run. But alas, they often are not, so it often does not. Engaging in literary dialectics strengthens the skills we need to repair buggy programs.

But perhaps more importantly, viewing programs as arguments may also help us write fewer bugs in the first place and spot them in other code. In Gorgias, Plato trains us to make arguments and assess them critically[6]. He demonstrates that not all arguments work, even when the speaker is very clever or when the master seemingly smites an opponent. Plato shows us the difference between this kind of performance and a truly sound argument. But he doesn't point out where Socrates errs; we learn to recognize flawed assumptions by engaging in the mental gymnastics for which literature trains us. The practice of reading these texts improves our mental acuity and our ability to reason through arguments.

Note that the form matters; Socrates' dialogues raise unanswered questions, generating effects that a mere treatise or recitation of facts couldn't produce. In How to do Things with Fictions, Landy says, "Over and above teaching us, Plato's dialogues have the capacity to train us. If we have a predisposition for detecting and are interested in resolving conflicts within a position—if, that is, we instinctively posit logical consistency as a desideratum in life—then we stand to learn not only what to think, but also, and far more importantly, how to think.[7]"

---

[6] Joshua Landy. *Philosophical Training Grounds: Socratic Sophistry and Platonic Perfection in Symposium and Gorgias.* Arion 15 (1): 63-122, 2007.
[7] Joshua Landy. *How to Do Things with Fictions*. Oxford University Press, 2012.

If programs are like arguments, then to know how to make watertight arguments is to know something about writing functional programs. To know how to assess arguments is to know something about reviewing code. To know how to spot bad arguments and repair them is to know something about debugging. I hope by now you are having second thoughts about converting the Humanities Library into a laser tag venue where the CS Department and D-School can blow off some steam. But just in case, let's talk about authors.

Authors as Functions

Programmers who read literature have a more positive impact on the industry because they understand the relationships between an author and his work.

In What is an Author?, Foucault describes authors as "classifying functions[8]". When we read an author's work, the author's reputation mixes with the words on the page and influences how we interpret them. Foucault calls this a "function" in the input-output sense: a work "passes through" a sort of mental filter and comes out the other side transformed—with new meaning or importance attached. For Foucault, authors don't precede their work but are consumed by it, bound to it. Authors exist in order to bring about their work.

Foucault reframes readership too: what a reader knows about the author changes how they perceive his work. Would we look differently upon Shakespeare's plays if we learned he was definitely homosexual? What if we knew he did write King Lear, but did not write Hamlet, Romeo and Juliet, or Othello? How would that impact our opinion of King Lear or the relative worth of Othello? What if we learned Nietzsche was a coal miner? Would we regard his works as inferior if they were not the product of an eccentric, powerful thinker, but a commoner?

Many programmers like to think that none of this applies to us—the programmer is a product of her outputs, and her outputs should be judged without the stain or stamp of her previous work. But this can't be the whole story. The Stanford CS department grades submissions partially on "style." We see the trappings of authorship-associated clout all around. We lean heavily on a programmer's open source projects to judge her merit. We secondarily note her followers on Github, her Hacker News karma, and her Stack Overflow points. These are all tracking systems for measuring participation in online venues where we expect good programmers to be active. In short, we resist the anonymity of the author-programmer in all kinds of bizarre and inventive ways.

Foucault's views on how an author's aura shapes the reception of their work is particularly important in light of how much time programmers spend reading code. Programmers read a lot: code, API specs, design docs, articles about code, etc. Like

---

[8] Foucault, Michel, and Paul Rabinow. *What is an Author?* The Foucault Reader. New York: Pantheon Books, 1984.

readers of literature, programmers connect the name of the author to the work as they review it. Their prior opinions of the author bleed into and transform their view of the code. Most of the time, programmers are reading as a means to an end: to green light a change, to fix something, or to find structures to mimic in their own work. This means that the biases we develop about authors impact our work and the work of others. We go into reviews with either scrutiny or amnesty on our minds and, most of the time, we have our biases confirmed. As a result, these so-called 'confirmation biases' also affect what code goes into production.

As elsewhere, confirmation bias tends to bury some good ideas and give a free pass to some bad ones. Biases exist everywhere, and if you apply the terms author and work broadly, we can find author functions adding value and wreaking havoc everywhere, but this is a particularly pernicious problem for programmers because we believe that we're immune. Of course, we're not.

One notorious example is the "Rubyist" (an early contributor to the Ruby programming language) who went by the pseudonym _why. He wrote prolifically and frenetically—often producing scraps very like those of Sappho that were scooped up by the community and cherished. In 2009, he committed "infosuicide," deleting his work from open source platforms, closing all of his accounts, and withdrawing from the Internet in a Kafka-esque attempt to kill the author[9]. Speculations on his motives abound. Disenchantment with the personality cult that buoyed his work based on his celebrity status certainly played a role in his choice to withdraw. Six years later, programmers still struggle to let go of his essence, even if holding on means holding back progress, "It took me a long time to become okay with removing thousands of lines of code that _why wrote," said Steve Klabnik who now runs _why's project HacketyHack. "WhyDay" is still celebrated on the anniversary of his infosuicide. It's a day meant to encourage hackers to "devote a day to doing fun, imperfect, creative projects in the spirit of _why.[10]"

In short, Foucault's concerns about authorship are very relevant in our community, where anonymity is a unicorn that can't be caught. The danger lies in our not knowing, or not accepting, that we are susceptible to confirmation bias. Our industry regards itself as highly meritocratic. We believe that work is judged by objective and technical standards. We believe that work is evaluated in the limit approaching isolation—independent of an author's previous work. We believe all of this despite the bizarre and multifarious ways that we've invented for attaching clout to individuals. We work in ecosystems where we can see the number of lines of code that each individual has contributed to a project, and yet most programmers would reject the claim that we focus on anything but technical aspects of the code we review. Our tools are set up to support this stance. We review "diffs," which highlight changes, displaying additions in green and subtractions in red. Like Proust taught us to do, subtracting the red from the green allows us to easily identify the individual's contribution. Our infrastructure gives us the sense that we are meritocratic. Our tools and rhetoric feel like the trappings of a meritocracy.

---

[9] Roland Barthes. *Death of the author*. Aspen, no. 5-6. 1967.
[10] Annie Lowrey. *Where's _why?.* Slate Magazine. March 15, 2012.

Yet it simply isn't the case that our code is reviewed in isolation from the coder. If we read code written by a senior engineer, mistakenly thinking that it was written by a recent hire (because both authors were editing the same file), we regard that code differently than we would if we knew and trusted the author. We may think it our duty to do so. Authorship certainly impacts the way we review code. After repeated review of someone's work, we begin to form a sense of who they are as a programmer. The work of an experienced programmer gets an a priori boost from their name alone; their code is treated with deference, scrutinized lightly, if at all.

This isn't altogether bad. There are reasons these biases exist. But we can already see the relevance of Foucault's authorship to programming. What is an Author? is a meta conversation on the subject of authorship, but all literature trains us to be aware of this relationship. The ramifications of author functions are very pronounced in literature. We treasure even Sappho's scraps and fragments and try to reconstruct the missing parts based on everything we know about how she saw the world. We likely would not do this for an author whose function had not brought a canon of cherished poems into the world.

By reading literature, we learn that author functions are ubiquitous; none of us are immune. In reading literature and in reviewing code, we may have sound reasons for treating authors differently. But literature teaches us to at least acknowledge and evaluate these biases. By failing to do so, we affect our colleagues and the code that goes into production without accounting for our own reasoning. Literature trains us to be cognizant of this important relationship, which allows us to interact with our work and our colleagues' work in ways that are deliberate and defensible.

Maybe we should be entirely meritocratic, ignoring an author's past oeuvre. Maybe, by contrast, we should allow a programmer's existing corpus of work to impact how we think about their future offerings. Or maybe the best approach is somewhere in between: to embrace the human element of software development while still being equitable in our dealings with colleagues, all while making sure that the best code is in production. Regardless, our best chance of achieving any of these utopias lies in an awareness of our biases that allows us to act deliberately.

Conclusion

I'll end where I started, by saying that there are many reasons to believe that studying literature is a vital practice. The humanities are and have always been subversive. They investigate every claim and leave no rock unturned. They resist false authority regardless of its origin. They are the cradle and the incubator of skepticism, and they nurture that skepticism in the face of questions big and broad enough to make many flee from their shadow.

I agree with all of these arguments about literature and about the importance of the humanities more generally. I agree that private life, healthy society, and good citizenry are all areas where literature can have an important impact, but my focus here has been, instead, that the serious study of literature provides benefits that are

commercially important and skill-specific to programmers. These are not idiosyncratic claims[11], and there is ample data to support them.

To be sure—lest this argument make the goods of literature seem too ripe for easy plucking—the benefits discussed here are not automatic. Bad and lazy study of great literature will bring none of this gravy. Neither will the industrious study of bad literature. Rather, in all the ways mentioned, great literature is a training tool. It only provides us with the venue and tools that we need to develop these talents, not the motivation or drive we need to truly acquire them.

The Stanford Computer Science Department knows all of this, at least some version of it. The department is making strides to cultivate rigorous interdisciplinary work. The new CS+X joint major is one example of efforts aimed at integrating humanities and computer science coursework. Real synthesis is becoming a viable option for Stanford undergraduates. My faculty and industry mentors all agree that this stuff is vital; they've supported my literary, philosophical, and journalistic pursuits with no certainty that I would become a better coder because of it. We've been slow to defend our vague intuition that studying the humanities is good for us in terms that appeal to the frantic engineering undergrad's sense of urgency and contempt for the vague. But we're arriving at that moment. Choosing to axe literature lectures for engineers now is a tragic choice; we're just beginning to see real progress toward interdisciplinary work that can prove its own value. In the past, we've sometimes taken superficial, 'faith-based' approaches to interdisciplinary work. We've assumed it should help and so it will help, although we don't know how. This essay provides three ways this actually can and does work.

Reading literature transmits perspective. Reading literature helps us assess and assemble good arguments. Reading literature teaches us the ubiquity and impact of the relationship between an author and his work. Of course, these are not the only three. But if you want to gauge whether an interdisciplinary approach has merit for future software engineers, it might be worth asking whether it meets any or all of these criteria. The world doesn't actually need more nerds who can misquote Shakespeare. That's not the point. The point is that these three skills, which can be acquired through the serious study of literature, map directly onto improvements that are vital to the growth of a great programmer and to the production of great software.

---

[11] See the phenomenal blog humanitiesplus.byu.edu, which aggregates "reasons and strategies for enriching vocational training with skills provided by the Humanities."