Darwin’s Dangerous Idea

November 7, 2019

Darwin’s Dangerous Idea: Evolution and the Meanings of Life is a 1995 book by Daniel Dennett. At the risk of monumental understatement, this book is full of interesting ideas that will surely change the way readers think about a great many things. But since this is a software development blog, I’ll focus on just a few core elements of that book. The central idea is that natural selection is a blind, algorithmic process that is sufficient to “design” complex things (such as organisms) over time. After I read this book, the one takeaway that altered my thinking most profoundly was that the natural world makes much more sense if you think of its design as bottom-up, rather than top-down. Without getting too philosophical, the book contrasts the idea of a “God/higher intelligence” as the designer of the “universe” (top-down), as opposed to natural selection acting on simpler components and creating greater complexity (or more “design”) over time (bottom-up). Dennett’s arguments and discussion in the book are obviously orders of magnitude more detailed and thorough than this brief introduction, but the basic idea is relevant to the process of software development, or at least, provides some tidbits to explore.

In the past, I’ve praised Joel Spolsky’s blog about Architecture Astronauts. The reason I love that blog is because it was one of the first I read that clearly identifies the single biggest problem (by far) in software development. Hint: the problem has nothing to do with technical acumen, but everything to do with developer narcissism.

How does this relate to Darwin’s Dangerous Idea? It relates because the Astronauts start with a top-down design approach. They identify a bunch of “advanced” techniques that they want to use (e.g. DI containers, factories, polymorphism, TDD, ORM, Gang of Four design patterns, anything else new and shiny that is complicated to implement, or can be cited as evidence by the developer that he/she is really, really smart, etc.). Then, after laying out the grand design, they shoehorn all of the lower-level implementation details (i.e. the code that does the actual work, including that pesky database) into their towering architectural masterpiece, even where the (square) implementation details don’t fit into the (round) architectural gospel.

To be clear, some “advanced” techniques may be warranted by the needs of a project, and then, of course, it is appropriate to implement them. But never as a starting point. Let me give you an example that came up recently. Over the past month or so, I’ve been learning React. In that course of that effort, I’ve also been investigating Redux since those two things are often used together. I discovered a blog by Dan Abramov (co-author of Redux) entitled You Might Not Need Redux. Dan points out that using Redux involves tradeoffs, and that you may not want to introduce the additional complexity of Redux into your app if you don’t really need it.

Eureka! The point is: often times, developers rush into implementing a coding technique, or a library, etc. that isn't really necessary, and it adds significant, unnecessary complexity. Over my 20+ years in software development, I've seen this problem play out time and time again. In fact, I've often had the unfortunate task of trying to scale back/refactor code to just what's necessary. And, as I've noted in the past, it is much, much more difficult to scale back complexity than it is to introduce more complexity when it becomes necessary.

I've heard people criticize simple architectures because, the argument goes, they won't be able to handle more complex situations (e.g. new unforeseen modules need to be added, number of users grows, etc.). Nothing could be further from the truth. If you start with a simple, but solid, foundation, adding complexity where necessary is normally not a problem. And this, in the parlance of Darwin's Dangerous Idea, is exactly how "Nature" designs. It starts simple and then adds complexity to the foundation over time, but only as required by the natural environment. Of course, this analogy isn't perfect. The point though, is that it is much more sensible to start with simple fundamentals and then build upon those as necessary. That way, software is only as complex as it needs to be. Needless to say, this has enormous benefits, particularly since the lion's share of time will ultimately be spent debugging and maintaining the code base.

The foregoing discussion is one of the reasons why I've always been a strong proponent of a data-first approach (as opposed to code/business objects first). Data sits at a lower (or perhaps the lowest) point in the "evolutionary chain" (for lack of a better term) of a software project. You have to get the data right. If you don't, everything else is a waste of time (or, at least, you'll end up wasting a lot of time).

Finally, let's generalize a bit further while we're here. I don't have a CS degree, but I took several college-level programming courses prior to starting my career. I was initially surprised by the disconnect between those courses and how software is actually developed (or should be developed) in the real world. Of course, to be fair, this can be said about many disciplines. But in software, in particular, this is undoubtedly a major component of the problem; people clinging to that theoretical bias, whether it be to (somehow) validate their education, or to prove their superiority.

I'm reminded of that great quote (attributed to multiple people): "In theory, there is no difference between theory and practice. But, in practice, there is."