Abstractions

September 24, 2018

Over the course of 20+ years in software development, I've probably worked with over 200 developers. Some of them were outstanding; others not so much. If I had to identify what makes a developer great, I could cite many factors: experience, intelligence, problem-solving skills, etc., but there's another critical factor that is less obvious – choosing the appropriate level of abstraction. In fact, I can usually gauge immediately the caliber of the developers I'm working with based on this single factor.* What do I mean? Let's start with a simple example.

One of my first jobs involved developing business apps for a publishing company. I was a Visual Basic developer and one of the other guys there was a C++ developer. This particular guy was an excellent developer, but it took him three times as long to put an app together than I could do in VB. One of the reasons is because VB abstracts out a lot of the low-level crap that C++ developers have to deal with (e.g. memory management). For 99% of business applications, VB worked just fine, and even VB's detractors would have to concede that it revolutionized the development of business applications. The point is that VB abstracted out most of the unnecessary low-level code that has no effect on how the application performs its functions. Therefore, it was the right choice for the vast majority of business applications.

Let's consider another example. There's been a trend over the last several years to use Object Relational Mapping (ORM), which abstracts out much of the code needed to deal with the database. For example, using ORM, you don't have to write nearly as many SQL queries; the ORM engine generates those for you. Unlike the VB example, however, ORM is a case where abstraction is a bad thing because it is crucial to have lower-level control of the database. Early in my career, I attended a Microsoft seminar on writing high-performance applications. I'll never forget it. One of the first things the speaker said was: "Performance is in your database." He then proceeded to hash that statement out, and I realized that he was right on. The database (and data access code) has to be optimized; otherwise you're asking for performance problems. See my previous blog for a more thorough discussion of ORM ("ORM is an Anti-Pattern"). The point again is: knowing when to abstract something and when not to.

The final area I'd like to discuss is object-oriented program. I've spent many years involved in OOP at various levels of complexity. I worked at one company where virtually everything was an abstraction and everything was coded to interfaces. You've probably heard the old saw that you should "code to interfaces and not to implementations." This general concept has some theoretical heft; however, it can get absurd rather quickly. One example is the idea that a data access layer should be abstracted so that you could swap out the database from one vendor to another, and the calling code wouldn't have to change. You've probably heard someone say

that, or at least you've read a software development book that touts this idea. While I've certainly seen people "upgrade" their database in new projects from say, Oracle to SQL Server, I've never seen anyone decide midstream to switch database platforms. The bottom line is that I would never create a set of abstractions (interfaces) in order to provide this flexibility even though some astronaut thinks that this is good "engineering".

The other killer that I've seen destroy applications is the overuse of inheritance, and more generally the over-engineering of OOP. There's a lot of spaghetti code out there. Indeed, I suspect that the majority of codebases could be accurately described as spaghetti. But all spaghetti is not created equal. The worst tasting spaghetti by far is OO spaghetti. Why does this happen? Usually, OO spaghetti begins when some developer decides to implement an architecture that he/she either learned from a book, or was influenced by a college programming course. Or perhaps they've just been seduced by conventional wisdom and can't think outside of the box. Whatever the genesis; over time, it is nearly impossible to maintain a theoretically pure OO structure for many reasons, not the least of which is that the complexity is unsustainable on almost all fronts: maintainability, debuggability, scalability, etc.) In addition, most hardcore OO solutions for business applications don't extend well, despite the fact that extensibility is touted as one of OOP's strengths. I can't tell you how many times I've had the unpleasant experience of trying to shoehorn some new code into a mess of unnecessary inheritance, factories, DI containers, etc.

This discussion reminds me of Daniel Lemire's wonderful blog "Simplistic Programming is Underrated". I suggest you check it out.

---

* Obviously, characterizing the caliber of a developer is a somewhat subjective exercise. My standards are writing clean, maintainable, debuggable code that solves the problems at hand in the simplest way possible. Developers obsessed with over-architected solutions are consistently the worst measured by these standards. Not to mention the fact that they are not enjoyable to work with, despite their "sophistication."