Code antifragility

Antifragile applications

The term *antifragile* is coined by Nassim N. Taleb, in his eponymous book. Fragile things and beings suffer from a little external stress. Robust things and beings are mostly inert to it. Antifragile things and beings benefit from low levels of external stress. They are much better at withstanding shocks. (This document is written in Covid-19 time.)

The nature of a larger software application

There is an awful amount of boiler-plate code in run-of-the-mill software applications. But it is not because there are many pretty standard aspects to it, that the application is unimportant, without value, etc. It will undergo the same lifecycle of maintenance being much more expensive than development, and, over the span of time, multiple partial rewrites, potentially in other programming languages.

Secondly, there is inherent layering present in such applications: one can find modules, sub-modules, components, etc. There are many reasons to believe that such larger applications behave like a complex system, with many dependencies between components, explicitly visible but much more frequently, hidden.

Motivation

The company intends to be big in making source code (and therefore, the applications described by them) *antifragile*, the exact opposite of fragile. I envisage a system of actively swapping out parts of the application by equivalent ones; which will allow the whole application to morph over time.

All begins with a system of code duplication detection. Say a small sub-routine or class is used to perform the sorting of objects. Sorting is very standard, taught to most computer science undergrads, pretty quick to implement. Hundreds of variations exist.

My application was using the standard Java `Collections.sort()` library method at the time. At some point the JDK swapped implementations (from some sort of merge sort to TimSort), which caused crashes somewhere deep in my application. It took days to remove them: it turned out that I had not been following the protocol properly. Swapping implementations as part of a testing procedure would have helped.

There are quite a few variations on sorting which are less readily available. At some point I had to implement one where next to the sorting a very large array, I also had to maintain an index to the elements to be sorted. When faced with this task, I was stuck by

1. the fact that someone should have implemented this before, but it would take a while to find such an implementation, and then test it sufficiently to trust it.
2. the point that it would be really beneficial, both for my own edification as for those who had to maintain the code later, to have our own minimal implementation available.

The first point highlights that there are thousands of interesting software constructs to be used in an application. No software engineer can be expected to remember all the names. *Implementing the construct is probably the fastest way of finding it.*

The second point builds on the first: if you have to write the code to explain what it is that you need, you have to have the expertise to do so. Reading implementations in the style that you are used to, i.e., compatible with the rest of the code, is much easier than reading code in some foreign library.

But a code duplication detection system that has many external libraries and my own application, should be able to find commonality, and, if sufficiently advanced, may notify me of a duplicate. There are many positive things about this.

1. Sometimes the duplicate is in your own code. You can do without this.
2. The duplicate may be in a library. Its implementation may be better than yours, in many different ways: yours has a bug, forgets to check some boundary condition, is less documented, is slower, etc.
3. The duplication in the library may actually be inferior to yours! Maybe there is a bug or problem in the library. Other parts of your code could do with using your implementation rather than the library.
4. Yours will very likely be functionally identical, *for your purpose*. Still, this has the advantage of being more readable, because it has been written it the common style of the application.

Imagine that we can identify common constructs and sub-routines in your application, and purposely swap them out with other implementations. Which advantages would this bring?

1. it helps stress testing. You may have been relying on specific implementation quirks.
2. over time, libraries evolve at a different speed than your application and other libraries that you are using. The interconnected network may have to be shifted; other implementations may have to be plugged in.
3. when it is time for a now implementation of your application in a more modern language, you'll get quite a few for free
4. it would identify parts of your code that are really application specific. No idea at this point how big that part would be, but it would not surprise me if this was less than 70% for many apps.