# Software Finds Its Way

Another week, another pair of large maintenance tasks. The first task was to do a full replacement of a logging system. This logger was built over the span of 10 years, was a bit “crufty,” and lacked some functionality we needed. The other project was/is a bit larger and we are code naming this one: “Changing the engine while the truck is driving down the road.“ I’ll talk about both of these below.

When this application (The Truck) was built, its sole purpose was to maintain a client-facing portal. This included building mundane features like user and group management, menu management, entitlements, and some other minor features. The client-facing portal eventually went away, but the admin portal stayed. It was extended to manage data warehouses, data flow processes, logging, monitoring, and some reporting. This acorn of a software project became a full featured oak tree of an application.

Here's a fact: You will spend more time in maintenance than in development.

When I was new to this business, it was drilled into me that the real cost of software was in the maintenance, not the initial construction. There are some studies that attempt to quantify the cost of maintenance vs. development. The article: “Which Factors Affect Software Maintenance Costs More?” from the National Library of Medicine (<https://pmc.ncbi.nlm.nih.gov/articles/PMC3610582/>), mentions a number as high as 90%. It’s my experience that this number doesn’t seem entirely unreasonable.

Sometimes software maintenance requires a new direction. While working on a compiler issue in a .NET MAUI application we built, I raised an internal red flag on this go-around. It seems that nearly every month, we’d been fighting compiler issues because of some change made to one part or another of our development stack. Potential issue vectors included Visual Studio, .NET Core, .NET MAUI, and a few libraries. It seems like every month, we faced some complier issue or another and wasted an inordinate amount of time just getting our code to build.

What was the source of this? In all reality, it came from how we used the Roslyn compiler. We used a designer to build forms. This designer calls a code generator, which builds and compiles a generated Model-View-ViewModel (MVVM) in the runtime. Last weekend, I wasn’t looking forward to another day in compiler land, so I decided to rethink how we implemented it and did some experiments to see if a new direction was viable. After a couple of hours on the weekend and a full week day, I realized that a new direction was 100% doable. But would a new direction have any added benefits or just be a lateral move? As I coded up the new direction, I began to see easier solutions to thorny problems that remained, as well as improvements to current processes that this change would enable.

The theme among these items is this: Software is a long game and can be expensive to maintain.

As you’ll recall, one project was to overhaul the logging code in our admin portal. How did we do this in a live running system? The first task was to build the actual logger itself, including the viewer. Once we were happy with our code/design, we wired it into a few places that see a lot of logging activity. We let it cook for a few weeks, optimizing the logger as well as the viewer. Having real-world use helped flesh out the features with low risk to our users.

I think using polymorphism is one of the most useful techniques for lowering long-term maintenance costs. Writing to an interface can lower costs when that system may need updating or replacement. The .NET MAUI application I mentioned above needed new directions because of the constant compiler issues. The question was: Can we replace the compiler part of the application with something else? If so, how much effort would it take?

Luckily for us, this application constructs two different artifacts: a View, which is bound to properties on a View Model. Could we just swap a new type of View Model in place of the old one? After a half-day of work, the proof of concept worked and we re-implemented a few more critical features to solidify the viability of the project. After proving its viability, we made the GO decision to complete the swap.

One of my all-time favorite features of .NET is the concept of overloads. You can have a function with the same name and varying argument signatures. This is very useful in maintaining code. Function arguments are one of those “don’t know what you don’t know” kind of maintenance issues. Your initial pass at a function may not have the proper parameters needed, or a new requirement might need a bit more information. Bring in the wrapper. The code snippet below demonstrates this:

public void ReloadVendorFiles(Packet

 launchPacket)

{

ReloadVendorFiles(

 launchPacket.StartDate,

 launchPacket.EndDate,

 launchPacket.LoadType,

 launchPacket.Database);

}

public void ReloadVendorFiles(

 string startDateRange,

 string endDateRange,

 string loadType,

 string database = "THE\_OTHER\_DATABASE")

This technique involves creating a version of your function that requires additional arguments and wrapping that call inside of a simpler function, normally with lesser arguments.

With the cost of maintenance being so high, it makes sense to use techniques to help lower costs whenever possible. The techniques (new direction, pilot code, polymorphism, and wrapping/overloading functions) above are just the tip of the iceberg in making software easier to maintain. What techniques do you use?