VBA+OOP: What, When, Why

Posted on August 29, 2018August 29, 2018 by Rubberduck VBA
As I'm writing a series of articles about a full-blown OOP Battleship game (https://rubberduckvba.wordpress.com/2018/08/28/oop-battleship-part-1-the-patterns/), and generally speaking keep babbling about OOP in VBA all the time, it occurred to me that I might have failed to clearly address *when* OOP is a good thing in VBA.

OOP is a *paradigm*, which entails a specific *way of thinking about code*. Functional Programming (FP) is another paradigm, which entails another different way of thinking about code. Procedural Programming is also a paradigm – one where code is essentially a *sequence of executable statements*. Each paradigm has its pros and cons; each paradigm has value, a set of problems that are particularly well-adapted to it, …and its flock of religious zealots that swear they saw the Truth and that their way is The One True Way: don't believe everything you read on the Internet – think of one (doesn't matter which) as a hammer, another as a screwdriver, and the other as a shovel.

Don't be on the "Team Hammer!" or "Team Screwdriver!", or "Team Shovel!" – the whole (sometimes heated) debate around OOP vs FP is a false dichotomy. Different tools work best for different jobs.

So the first question you need to ask yourself is…

What are you using VBA for?

Scripting/Automation

If you're merely *scripting* Excel automation, you very likely don't need OOP. An object-oriented approach to scripting makes no sense, feels bloated, and way, way overkill. Don't go there. Instead, a possible approach to *cleaner code* could be to write one macro per module: have a `Public` procedure at the top (your "entry point"), at a high *abstraction level* so it's easy to tell at a glance everything it does – then have all the `Private` procedures it calls underneath, listed in the order they're invoked, so that the module/macro essentially unfolds like a story, with the high-level bird's eye view at the top, and the low-level gory details at the bottom.

The concept at play here is *abstraction levels* – see *abstraction* is one of the pillars of OOP, but it's not *inherently* OOP. Abstraction is a very good thing to have in plain procedural code too!

Procedural Programming isn't inherently bad, nor evil. *Well-written* procedural code at the right abstraction level is a pleasure to read, and when you think in terms of *functions* (i.e. inputs -> output) rather than "steps" or "instructions", then you can write *pure functions* – and pure functions can (and probably should) be unit-tested, too.

If you've ever written a *User-Defined Function* that a worksheet invokes, you've likely written a *pure function*: it takes input, and produces output *without accessing or altering any other state*. If you've done that, congratulations, you've learned the fundamental building block of the *Functional Programming* paradigm! ..then again, pure functions aren't *inherently* FP.

The vast majority of VBA code written, falls in this category. It would likely be *toxic* to try to squeeze OOP into such code; I'll even say that OOP is flat-out the wrong approach here. However, an FP-like approach isn't necessarily a bad idea… although, VBA clearly wasn't designed with *Functional Programming* in mind, so you'll hit the language's limitations very early in the process… but it can't hurt to design your script avoiding side-effecting functions and proliferating global state.

Framework/Toolbox Code

Somewhere in-between the *script* and the full-blown *application*, there's this type of VBA project that you write for yourself as some kind of "toolbox" with all kinds of useful code that you often carry around and pretty much systematically import into every one of your new VBA projects.

This, in my opinion, is where OOP really shines the brightest in VBA: it doesn't matter if it's *procedural programming* code consuming these objects – it's OOP nonetheless. As much as the Excel object model itself is made of objects, and couldn't care less if it's procedural or object-oriented code consuming it.

We could be talking about a fully-reusable ProgressIndicator (https://rubberduckvba.wordpress.com/2018/01/12/progress-indicator/) class, some polymorphic `Logger` tool that the consuming code can configure as needed to log to the debugger, some text file, or a database, or a set of custom data type classes – a `Stack`, or an `ArrayList` wrapper, or a `File` class that wraps file I/O operations and maybe some `Scripting.FileSystemObject` functionality, or something else: you get the idea.

Full-Blown Applications

If you're seeing VBA as a document-hosted VB6 (it pretty much literally *is*) that can do everything a VB6 program can do, then you're looking at something else entirely – and the problems you're solving are in a completely different realm: you're not automating spreadsheets anymore: you're writing a CRUD application to automate or facilitate data entry into your ERP system, or you're maintaining a set of support tables in some corporate database, …likely, something a programmer would look at and ask "hey why are you doing this in VBA?"

"Because I can" is a perfectly acceptable answer here, although "because I *have to*" is often more likely. Regardless, it doesn't matter: well-written VBA code is better than poorly-written VB.NET or C# code (or Java, or anything else): if you're writing VB.NET and it says "Imports Microsoft.VisualBasic" at the top of your modules/classes, then you're likely not writing idiomatic .NET code, you're writing glorified VB6 using modern syntax, in a modern IDE.

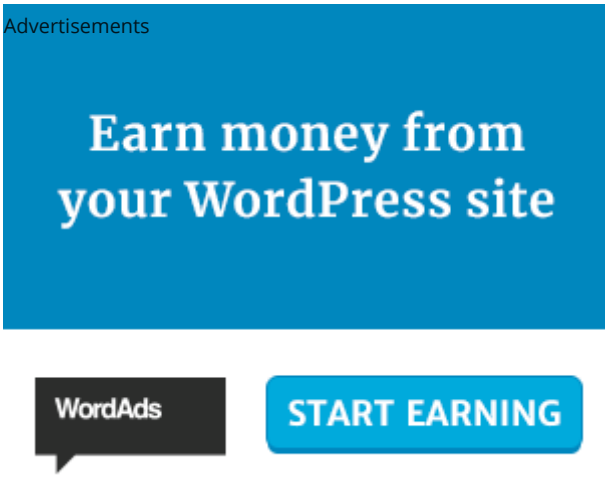
Bad code is on the programmer, not the language.

When you're making an *application*, procedural programming can be actively harmful – you're building a complex system, using a paradigm that doesn't *scale* well. FP would be an option for the bulk of the application logic, but then again VBA wasn't made for *Functional Programming*. An Object-Oriented approach seems the most sensible option here.

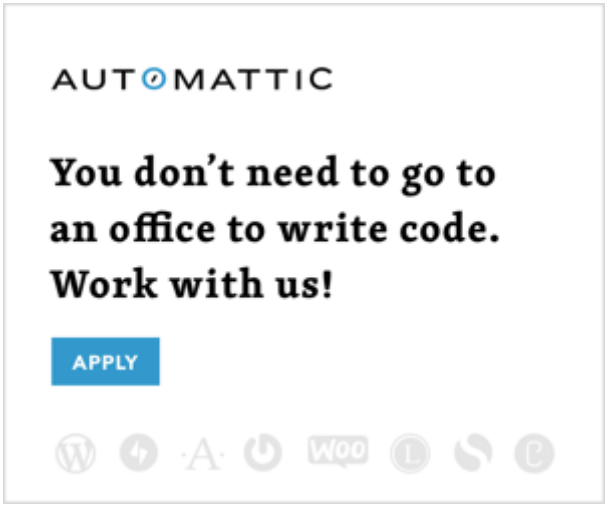
But what about RAD?

Rapid Application Development software, such as Microsoft Access, blurs the lines: now you're given a framework to write *event-driven* code (which *does* stem from OOP), but using object-oriented patterns (e.g. MVC) *can* feel like you're working *against* that framework… which is never a good sign. The best approach here would be to embrace the framework, *and* to extract as much of the logic as possible into small/specialized, self-contained components that can be individually tested.

Advertisements



REPORT THIS AD



REPORT THIS AD

Posted in OOP, tutorials, unit-testing, vba*Tagged* design-patterns, oop, tutorial, vba

Published by Rubberduck VBA

I'm Mathieu Guindon (Microsoft MVP Office Apps & Services, 2018), you may have known me as "Mat's Mug" on Stack Overflow and Code Review Stack Exchange. I manage the Rubberduck open-source project, whose goal is to bring the Visual Basic Editor (VBE) - VBA's IDE - into the 21st century, by providing features modern IDE's provide. *View all posts by Rubberduck VBA*