OOP in VBA?

Posted on December 24, 2015January 20, 2016 by Rubberduck VBA

VBA is an *Object-Oriented* language…

…whether we agree or not.

Object-Oriented Programming (OOP) is really all about 4 little things:

- **Abstraction**.
- **Encapsulation**.
- **Polymorphism**.
- **Inheritance**.

To make things clear: **there's no inheritance in VBA**. *But it doesn't matter*, because we can easily compensate with *composition*, which is often a better design decision, even in languages that support class inheritance.

The key to OOP, is *classes*. Why? Because classes are a blueprint for *objects*, …which are *kinda* the whole point of OOP.

Abstraction

If you've been writing code, you've been making *abstractions*. A procedure is *abstracting* a series of executable operations; a module abstracts a group of related operations, even variables are an abstraction, abstracting the result of an operation.

Or is that too abstract?

Levels of abstraction

If you think of the steps required to, say, make coffee, you might think of something like this:

- Make sure there's water in the coffee maker
- Make sure there's coffee in the coffee maker
- Start the coffee maker

That would certainly make coffee, right?

What sub-steps could there be to *make sure there's water in the coffee maker*? And to *make sure there's coffee in the coffee maker*? Or even to *start the coffee maker*? These sub-steps are *at a lower level of abstraction* than the 3 higher-level ones.

Clean code operates on a single level of abstraction, and calls into more and more specialized code: notice we don't care where the water compartment is at the higher levels.

That's why we put the public members at the top: because they're at a higher level of abstraction than the private members they're calling.

Classes are an important abstraction: they define *objects*, which *encapsulate* data and expose methods to operate on it.

Encapsulation

Similar to abstraction, encapsulation abstracts away implementation details, exposing only what *other code* needs to work with.

Global variables are pretty much the opposite of encapsulation; and if you have a public field in a class module, you're not *encapsulating* your data.

Instead of exposing a field, you'll be exposing *properties*. Property accessors can have logic in them, and that's the beauty of encapsulation: you're keeping a value to yourself, and telling the rest of the world only what it needs to know.

Polymorphism

If you've never worked with interfaces before, that one can be hard to grasp… but it's the coolest thing to do in VBA, because it truly unlocks the OOP-ness of the language.

Once, I implemented *IRepository* and *IUnitOfWork* interfaces in VBA. These interfaces allowed me to run my code using "fake" repositories and a "mock" unit of work, so I was able to develop a little CRUD application in Excel VBA, and test every single bit of functionality, without ever actually connecting to a database.

That worked, because I wrote the code specifically to *depend on abstractions* – an *interface* is a wonderful abstraction. The code needed *something* that had the CRUD methods needed to operate on the database tables: it didn't care whether that *thing* used table A or table B – that's an implementation detail!

The ability of an object to take many forms, is called *polymorphism*. When code works against an *IRepository* object rather than a *CustomerRepository*, it doesn't matter that the concrete implementation is actually a *ProductRepository* or a *CollectionBasedTestRepository.*

Inheritance

VBA doens't have that, which is sometimes frustrating: the ability for a class to *inherit* members from another class – when two classes relate to each other in an "is-a" manner, inheritance is at play.

Yes, inheritance *is* one of the 4 pillars of OOP, and *composition* isn't. But inheritance has its pros and cons, and in many situations composition has more pros than cons. Well, *class inheritance* at least, but in VBA *class* and *interface* inheritance would be intertwined anyway, because a VBA interface is nothing more than a class with empty members.

What of *Composition*?

In VBA instead of saying that a class "is-a" something, we'll say that the class "has-a" something. Subtle, but important difference: most languages that *do* support inheritance only ever allow a given type to inherit from one, single class.

When an object encapsulates instances of other objects, it's leveraging *composition*. If you want, you can expose each member of the encapsulated object, and completely *simulate* class inheritance.

Ok…

…So, what does that have to do with Rubberduck?

Everything. The Visual Basic Editor (VBE) isn't really helping you to write Object-Oriented code. In fact, it's almost encouraging you *not* to.

Think of it:

The only way to **find an identifier** in a project is to make a text search and **iterate the results one by one**, including the false results.

The more classes and modules you have, **the harder organizing your project becomes**. And when you realize you need some sort of naming scheme to more efficiently find something in the alphabetically-sorted *Project Explorer*, **it's too late to rename anything** without breaking everything.

So people minimized the number of modules in their VBA projects, and wrote procedural code that can't quite be tested because of the *tight coupling* and *low cohesion*.

Tested?

I don't mean F5-debug "tested"; I mean automated tests that run a function 15 times with different input, tests that execute every line of application logic without popping a UI, hitting a database or the file system; tests that test *one thing*, **tests that *document what the code is supposed to be doing***, tests that fail when the code changes and breaks existing functionality you thought was totally unrelated.

Rubberduck loves OOP

It was already the case when the current v1.4.3 release was published, and the upcoming v2.0 release is going to confirm it: Rubberduck is a tool that helps you refactor legacy VBA code to OOP, and helps you write testable – **and tested** – object-oriented VBA code.

The *Find implementations* feature is but an example of a wonderful object-oriented navigation tool: it locates and lets you browse all classes that implement a given interface. Or all members, wherever they are, that implement a given interface member.

Is OOP overkill for VBA? Sometimes. Depends what you need VBA for. But the IDE shouldn't be what makes you second-guess whether it's a good idea to push a language as far as it can go.

Posted in rubberduck, vbaTagged add-in, oop, refactoring, rubberduck, v2.0, vba, vba tools

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*

1 thought on “OOP in VBA?”

1. **Tom Hammarberg** *May 25, 2018* *Reply*

Thanks! You made my day! Fun! Looking for examples…