rubberduckvba.com

Style Guide

Extended Edition

VBA • VB6

Mathieu Guindon

June 2023

Foreword

Whether you're a veteran developer or just starting your journey in the field, Rubberduck (RD) proves itself to be an invaluable asset when working with VBA code.

Featuring over 100 inspections, RD excels in offering advice and proposing best practices for the creation of cleaner and more efficient code. While many of these inspections offer hints towards writing better code, others go a step further, assisting in the identification of errors even before you find yourself in the debugging process.

Yet, the inspections are merely the beginning of what RD has to offer. It boasts a plethora of other practical features like code formatting, indenting, and Unit Testing. As soon as you incorporate it into your development process, you'll find it hard to imagine working without it.

Paired with this comprehensive Rubberduck Style Guide, RD not only provides excellent suggestions, but it also explains the reasoning behind them. As you soak up and apply these guidelines, you're paving the way to a more refined development skill set

– Wayne Phillips, author of *vbWatchDog* and *twinBASIC*

About the Author

Mathieu Guindon started being interested in programming around age 12, avidly learning everything about BASIC 2.0 for the Commodore64. Throughout his teens he'd dabble with QBASIC and later with Visual Basic 4, then 5, and then VB6, all as a hobby. Left it all behind to work in the retail industry, where he eventually discovered he could automate much of theclericalg parts of hisoffice job (and that of colleagues) using VBA. Soon he learned to use the Excel object model to achieve everything that needed to be don.. Fast-forward a few more years to 2013, Mathieu got involved with the *Code Review* Stack Exchange community and eventually was elected as a moderator. By then Mathieu had already authored hundreds upon hundreds of answers on Stacks Overflow, and was starting to generate quite a lot of VBA content on the *Code Review* site. By the time Chris joined his efforts to make VBA a badge-worthy tag on the CR site, and for several years after that, Mathieu was all over the VBA tag on Stack Overflow, and eventually climbed to the tag's top-3 all-time contributors. As Rubberduck gained traction and credibility, one of its core contributors (also MV, unbeknownst to Mathieu at the time!)) nominated Mathieu as a Microsoft MVP, and the prestigious award was given in January 2018 as a recognition for his effort, and was renewed yearly until content creation burned him in 202 and he'd simply not apply for a renewal beyond 20221.Today Mathieu is a .net and SQL developer, consultant still programming as a hobby, but Mathieu is also a talented (more than he'll admit anyway) musician that is currently into harmonica but played guitar before and started learning some piano for fun not too long ago as well..

Special Thanks

Special thanks to Rubberduck contributors Max, Ben, Vogel, all the Andrews (+ThunderFrame), Brian, Iven, and everyone else – Wayne, and Carlos, and Rob, Stephen, and everyone that ever contributed to Rubberduck in any way, including by reporting any issues or suggesting any features. Thanks to every single one of our stargazers on GitHub, and everyone that blogged about or otherwise reviewed Rubberduck or some of its features; many many many thanks to the Ko-fi community and platform, but more importantly my Ko-fi supporters and Rubberduck fans getting themselves Rubberduck swag through the Ko-fi shop and helping spread the duck everywhere VBA is! Thanks to the Stack Overflow and Code Review Stack Exchange communities for the early exposure and constant VBA content and inspection ideas.

And thank you Chris, for sparking all of this and inspiring the best possible name for an IDE add-in project.

This document couldn't have happened without the support of my wife and the curiosity of my children, either; thank you for all this time I know I can't take back.

What you are about to read is essentially the sum of everything I could think of that might contribute to help VBA developers learn new tricks and translatable skills that make you a better programmer regardless of what language you're working with. This material applies modern programming concepts and principles to a language that remained frozen in time 25 years ago. It may confront and directly contradict some advice or certain ideas you've learned from your own experiences or read about somewhere on the web. Admittedly everything I've built and written is also "somewhere on the web", but these guidelines and recommendations are mostly just modernizations largely inspired by what makes VB.NET code good VB.NET code that adheres to today's standards. The irony is that exactly none of this content leverages any language feature that wasn't already present 25 years ago.

Programming in VBA is full of exploding bear traps. Thankfully, Rubberduck inspections can avoid quite a large number of them, and if you're just beginning to learn and you're reading this document, you're in good hands: following these practices and using Rubberduck puts you a mile or two ahead of anyone else in the same spot without the tools you've just given yourself: a lot of the knowledge that went into some of these inspections took many years to gather, and encapsulate many years of the combined knowledge of several current and former experienced VBA developers and Microsoft MVP recipients and alumni.

If you're an experienced VBA developer discovering Rubberduck, we're about to rock your world.

The IDE enhancements Rubberduck brings to the table unlock a whole new level of working with a VBA project and can bring your workflow closer to that of a software developer – because whether we realize it or not, whether we want it or not, writing VBA code makes us programmers. Let's be proud of learning and using VBA!

Thank you, dear reader, for supporting what we're trying to do: getting VBA and the VBIDE into this century!

Table of Contents

About Rubberduck

Back in 2009, I was already a VBA veteran when I started learning C# and .NET; to me the VBIDE was everything I ever needed from an IDE, and I liked that it brought back fond memories from a decade earlier, when I was toying with VB6 and Visual Studio 6.0. And then I was given a license for Microsoft Visual Studio 2010 Ultimate with JetBrains' ReSharper (R#), and it changed *everything*.

Backstory

Navigating Visual Studio 2010 was a great deal easier than in the good old VBIDE. IntelliSense alone was such an improvement – not only you had parameter info, but now also xmldoc documentation that was parsed straight from comments in the code. Navigation tooling was completely elsewhere, too: my favorite R# feature quickly became the *Go to Anything* command, and refactoring capabilities completely changed how I approached mundane things such as *naming things* and cleaning up code.

For a long time, using Visual Studio without ReSharper felt like something was amiss: the add-in would do things the IDE needed to be doing, but wouldn't until many years later with the advent of Roslyn – a new compiler for .NET/C# that was completely written in C#: that's when Visual Studio started integrating all the features that made ReSharper a great tool.

I went on to pursue other challenges, ready to leave VBA behind me, but life had other plans.

By mid-2014 I was learning new things VBA could do I didn't even know about, and soon we were a merry bunch of crazy folks generating VBA content on the *Code Review Stack Exchange*[1] site, pushing the horizon of our collective VBA experience further with every new post. Soon we were implementing .NET concepts[2] in VBA, and as we explored the VBIDE Extensibility library we started toying with writing VBA code that could read, write, and invoke other VBA code; the trigger that ignited the project that would become Rubberduck, was a small Excel add-in that would find specific methods to execute, invoke them, and then a test outcome would be evaluated from Assert calls made inside the invoked procedure: porting this VBA unit testing library for Excel to C# and .NET would allow us to break free from VBA's constraints and extend the IDE instead of the host application, and making it work across all Office hosts was our first challenge, for we soon discovered that `Application.Run` wasn't a given.

I still knew nothing of *lexing* and *parsing* then, but I knew having programmatic access to all the code loaded in the editor meant we could conceivably write code that could understand VBA syntax, and then tell the user about things that regularly trip up beginners – for example if `Option Explicit` was missing from a module, it would be easy to tell, and report it in a custom IDE toolwindow.

[1] https://codereview.stackexchange.com
[2] BCL interfaces, some level of *reflection*, even (arguably) some level of dynamic delegates!

But that wasn't enough. I wanted Rubberduck to be the *ReSharper for VBA*, the de-facto add-in any serious VBA developer carries in their arsenal, the must-have free tool in the so-called *age of dawn* of VBA, if we dub the early 2000's its *golden age*. I wanted Rubberduck to *understand* VBA deeply enough for us to implement the holy grail of refactoring features: *extract method*. With R# you could select any number of lines anywhere within a scope, and with a keyboard shortcut you've just moved that chunk of code into its own nicely named separate procedure and replaced your selection with a call to the extracted method, and the tool automatically moved the variables that could move and guessed what the parameters and outputs should be. This, along with correctly renaming anything whatsoever, involves doing things you cannot do without a *meta* model of what's going on – any error in this model could mean the refactoring results in broken code, which would defeat its purpose (a *refactoring* operation does not change *what* the code does, only the means to its ends, the *how*).

What's funny is that there's now a whole generation of developers to whom such magical IDE features are just normal everyday features of *any* IDE, and they're right: in 2024 a great IDE delivers a great developer experience by easing navigation and guiding beginners without overwhelming them. Features are easy to discover and instinctive to use.

Except if they're the *macro guy* at the office working in the VBE. Then they're stuck in 1999, unless… unless they know about Rubberduck.

Rubberduck isn't without issues: as of v2.5.2 the add-in consumes *a lot* of resources and may not work properly with very large projects; starting up the add-in introduces a significant delay in loading the VBE, and COM integration comes with a certain number of caveats and potential issues that we didn't anticipate, such as the VBIDE's docking toolwindow mechanism. It may cause a bit of a lag, or it may not shutdown correctly and explode with an exception as the host application shuts down. But even with its issues, the VBIDE with Rubberduck is a much better experience than without.

Work has started on version 3.0, which will address most if not all the most serious 2.x issues by implementing a *language server* for the VBA language, thereby moving all the resource-hogging out of the host process. When it releases, Rubberduck 3.x will be everything we ever wanted it to be, and while various things will drastically change (we're introducing our own code editor!), the way VBA developers should be using Rubberduck and writing VBA code isn't going to – the guidelines and best practices outlined in this document are therefore applicable regardless of whether you're using Rubberduck 2.x, whether we're 3 years into the future and you're using Rubberduck 4.x, or whether you're not using Rubberduck at all.

Philosophy

One of the very first inspection to be implemented in Rubberduck was the `Option Explicit` inspection. Okay, part of it was just because it was a trivial one to implement even before we had an *actual* parser… but the basic idea was (and still is) that *nobody knows everything*, and it's with our *combined* knowledge that we make a mighty bunch, and that is why static code analysis in Rubberduck explains the reasoning behind each inspection result: there are quite many things Rubberduck warns of, that I had no idea about 10 or 15 years ago. That never stopped me (and won't stop you either) from writing VBA code that worked perfectly fine (except when it didn't), but whether we realize and accept it or not… a macro written in VBA code is a set of executable instructions, which makes it a *program*, which makes the act of writing it *programming*, HYPERLINK "https://rubberduckvba.wordpress.com/2019/07/11/im-not-a-programmer/"which makes us *programmers*, and we owe it to ourselves to commit to write quality code – not just code that *works*, but code that *reads* easily and that is a pleasure to navigate, maintain, and extend..

Being programmers that write and maintain VBA code does set us apart, mostly because the language isn't going anywhere, and the IDE is becoming more and more severely outdated and under-featured as years pass. Yet if the volume of VBA questions on Stack Overflow means anything, VBA is still very much alive, still very much being learned, and this is where Rubberduck and static code analysis comes in.

When I started learning about .NET and C# over a decade ago, there was this exciting new language feature they called LINQ for *Language-Integrated-Query* where you could start querying object collections pretty much literally like you would a database, and it was awesome (still is!). In order to make this possible, the C# compiler and the .NET framework and runtime itself had to undergo some very interesting changes Jon Skeet covers in details, but the point is… the new syntax was a bit off-putting at first, and came with new and important implications (closures, deferred execution), and the company I worked for gave us all a ReSharper license, and that is how and when I discovered that thorough & accurate static code analysis tooling could be a formidable educational tool.

And I want Rubberduck to be like that, to be the companion tool that looks at your code and reveals bits of trivia, hints like "*hey did you know this conditional assignment could be simplified?*", or "*if that condition was inverted you wouldn't need this empty block here*".

Maybe we don't agree about *Hungarian Notation*, and that's fine: Rubberduck wants you to be able to find it and rename it if that's what you want to do, but you can mute that particular inspection anytime. But I believe the tool *should* tell you what *Systems Hungarian* notation is when it calls it out, and perhaps it should even explain what *Apps Hungarian* is and give examples, because *Apps Hungarian* notation absolutely *is* useful and meaningful (think `o`-for-`OneBased`, or `src`-for-`Source` and `dst`-for-`Destination` prefixes). But `str`-for-`String`, `lng`-for-`Long`, **o**-for-`Object` (i.e., the intrinsic data types) is different: it says nothing about the *purpose* of the identifier.

Rubberduck flags obsolete code constructs and keywords, to `Global` declarations `On Local Error` statement,, explicit `Call` statements, `While...Wend` loops; all have no reason to exist in brand new, freshly-written VBA code, and quick-fixes can easily turn them into `Public` declarations, `On Error` statements, *implicit* `Call` statements (without the `Call` keyword!), an `Do While...Loop` structures.

Rubberduck wants to level up the VBE and nudge your programming towards objectively, *quantitatively* better code, and in 2024 it'll be a whole *decade* since it all started, and the project is celebrating with swag!

Figure 1 Swag shop item 1070110YK81 10Y celebration die-cut sticker (CAD $4.79)

Static Code Analysis

Among the first ideas to come to life, was having Rubberduck analyze the code it parses, and highlight potential issues. Many of them encourage objectively good coding practices to follow, others are more subjective observations that are debatable, but having the ability to scan a code base and point these out can help keep a *consistent* style. Because when subjectivity comes into play, all that really matters is that the style is *consistent*.

If you fire up Rubberduck on any legacy VBA project with any significant amount of code, there's a very high probability that static code analysis generates *tons* of inspection results for various mundane little things. Should your goal be to *quick-fix all the things* and have code that doesn't spawn *any* Rubberduck inspection results?

Perhaps surprisingly, the answer is a resounding "no", simply because an automated tool will be wrong, sometimes. The presence of diagnostics / inspection results indicates a *potential* issue with the code *as Rubberduck understands it*; if there's a flaw in that understanding, inspections might yield false positives, and/or quickfixes and refactorings may not be available or working as expected.

Code Metrics

Rubberduck could count the number of lines in a procedure and issue an inspection result when it's above a certain configurable threshold. In fact, things are slowly falling into place for it to eventually happen. But we wouldn't want you to just arbitrarily cut a procedure scope at 20 lines because an inspection said so! Rubberduck can measure line count, nesting levels, and *cyclomatic complexity*. These metrics can be used to identify problematic areas in a code base and methodically split up large complex problems into measurably much smaller and simpler ones.

Line Count simply counts the number of lines. Eventually this would expand into *Statements* and *Comments* counts, perhaps with percentages; 10% comments is probably considered a good sign, for example. But no tool is going to tell you that `'increments i` is a useless comment, and even the best tools would probably not tell the difference between a huge `'the following chunk of code does XYZ` banner comment and a *valuable* comment. Common wisdom is to keep this *line count metric down* as much as possible, but one should not do this at the expense of readability.

Nesting Levels counts the number of... well, nesting levels. While nesting two `For...Next` loops to iterate a 2D array (or a `Range` of cells) down and across is probably reasonable, further nesting is probably better off made implicit through a procedure call. Rule of thumb, it's always good idea to pull the body of a loop into its own parameterized procedure scope. *Arrow-shaped* code gets flattened, line count gets lower, and procedures become more specialized and have fewer reasons to fail that way.

Cyclomatic Complexity essentially calculates the number of independent execution paths in each procedure[3]. A procedure with a cyclomatic complexity above 5 is harder to follow than one with a complexity of 1 or 2, but it's not uncommon for a "God procedure" with nested loops and conditionals to measure in the high 40s or above.

The *code metrics* feature will eventually get all the attention it deserves (will probably end up backported from the RD3 repo), but as with inspections the general idea is to highlight procedures that could be harder to maintain than necessary, and nudge our users towards:

- Writing more, smaller, more specialized procedure scopes.
- Passing parameters between procedures instead of using global variables.
- Having more, smaller, more *cohesive* modules.

[3] https://en.wikipedia.org/wiki/Cyclomatic_complexity

Navigation Tools

You may or may not have noticed, but the Visual Basic Editor is nudging you in the *exact opposite* direction, because…

- Having fewer, larger, more general-purpose procedures puts you in a *scripting* mindset.
- Using globals instead of passing parameters around is perhaps a simpler thing to do.
- Having fewer, larger, more general-purpose modules makes it simpler to share the code between projects, and arguably easier to find things in the *Project Explorer*.

If you're writing a small script, you can *and probably should* absolutely do that.

But if you're like me then you've been pushing VBA to do things it wasn't really meant to do, and you're maintaining actual *applications* that could just as well be written in any other language out there, but you're doing it in VBA because [*your reasons are valid, whatever they are*].

And that's kind of a problem, because the VBE seems to actively *not* want you to write proper object-oriented code: its navigation tooling indeed makes it very hard to work in a project with *many* small modules, let alone an OOP project involving explicit interfaces and high abstraction levels.

Rubberduck lifts pretty much all the IDE limitations that hinder treating a VBA project as more than just an *automation script*. Now you can have a project with 135 class modules, all neatly organized by functionality into folders that can contain any module type, so a `UserForm` can appear right next to the classes that use it, without needing to resort to any kind of ugly prefixing schemes. You can right-click on an abstract interface (or one of its members) and quickly find all the classes that implement it. You get a *Find symbol* command that lets you quickly navigate to literally anything that has a name, anywhere in the project. Curious about the implementation details of a procedure, but don't want to break your flow by navigating to it? *Peek definition* takes you there without leaving where you're at.

Figure 2 Peek definition pops a floating panel conveniently showing the source code for the user-defined module or member you've selected.

Rubberduck can list all references to any given declaration (whether user-defined or not), and show them with their respective location and surrounding context. You can right-click any identifier and select "Find all references" from the Rubberduck menu, or you can place the caret on the identifier and click the "n references" button in the Rubberduck toolbar to achieve the same:

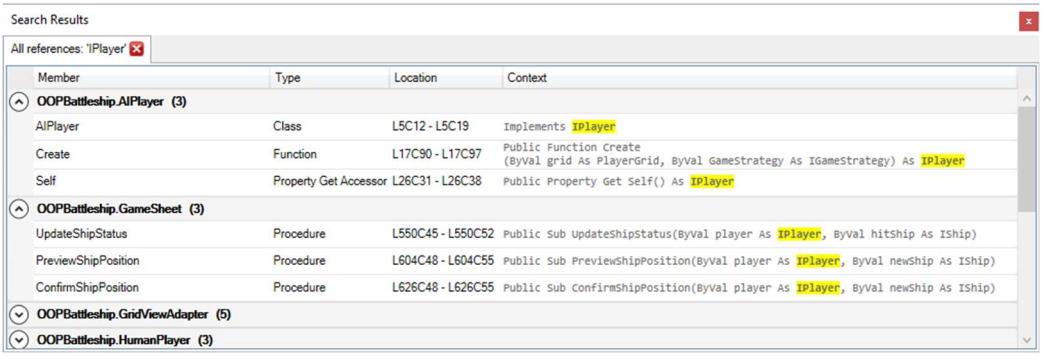

Figure 3 the Search Results toolwindow shows the "find all references" results.

Similarly, the decoupling that's achieved in advanced scenarios involving abstract interfaces means navigating to the definition of a method from one of its call sites (F2 in the VBE) takes you to the abstract interface member, so you'd have to search for the *Implements* statement that mentions the name of the interface to cycle through implementations – yikes! With Rubberduck you can just select "Find all implementations" to quickly get to the code you want to look at.

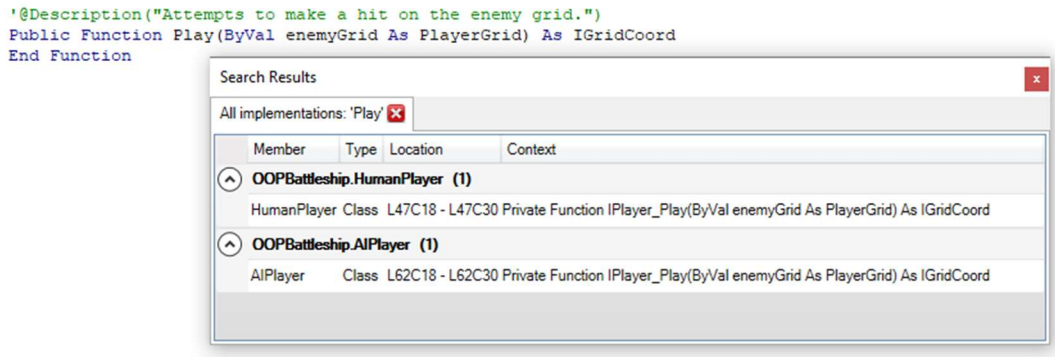

Figure 4 The Find all Implementations command is incredibly useful in object-oriented projects that leverage polymorphism through abstract interfaces: quickly locate and navigate to any implementation of any interface (class or member).

The VBE's *Project Explorer* aims to give you a bird's eye view of your project, regrouping modules by *module type* which is great for a small script that can get away with a small number of components, but that makes it very hard to manage larger projects.

With Rubberduck's *Code Explorer* you get to drill down to member level (and toggle showing/hiding signatures), and regroup modules by *functionality* using an entirely customizable folder hierarchy (folders are module metadata that you control using annotation

comments). Just that is more than enough to justify the feature, but over time *library* and *project references* got their own tree nodes, which makes much easier to see everything the project references at a glance… and then adding/removing these references is a massively enhanced experience too.

The tree view can be filtered with a search box, and you get to execute project, module, and member level commands without leaving where you're at, and selecting a node brings up its metadata (node type, identifier name, docstring/description, etc.).

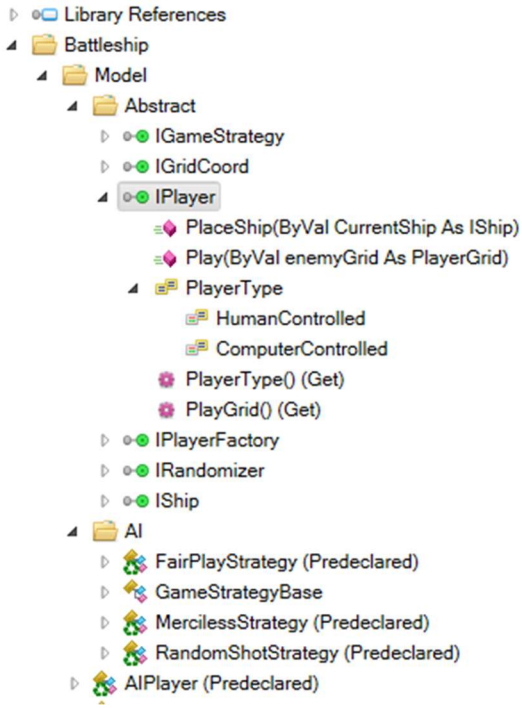

Figure 5 The Code Explorer leaves the VBE's Project Explorer in the dust, fair & square.

These navigational enhancements greatly simplify moving around a project of any size, although some of them might feel a bit *overkill* in a smaller project, and some of them are only useful in more advanced OOP scenarios. Still, having more than just a text-based search to look for things is very useful, and having the more advanced features means the IDE isn't going to be an obstacle to your learning.

Rubberduck is more than just a tool; it's an ode to the under-explored possibilities of VBA, it's a statement that says VBA is just as much of a "real" programming language as any other, and using it to its full potential means, indeed, a new iteration of what the best practices should be in VBA.

Guidelines

Guidelines

If there's one single over-arching principle guiding everything else, it would have to be ***write code that does what it says and says what it does***. Everything else seems to stem from this.

Many language features that were originally considered convenient are now obsolete: the `Def[Type]` statement alone demonstrates this obviously, but back when it was introduced it was probably useful to be able to implicitly type things based on the first character of their identifier name. Hungarian Notation prefixing schemes feel like they might stem from that era, as well as line numbering and the idea that variables in a given scope should all be declared at the very top of that scope.

The guiding principle for coding practices at the time all seem to be geared towards making code easier to write: implicit types, implicit modifiers, implicit type conversions, implicit member calls, short identifier names (they're even shorter without the vowels!)... meanwhile modern practices are all geared towards making code easier to *read*. Because programming involves writing code, yes, but it's mostly all about *reading* and *understanding* the code and removing ambiguities.

You may disagree with some of the recommendations enumerated here, and that's fine: these are ***warmly recommended guidelines***, not dogma. I will however present a reasoning for every one of them, regardless of subjectivity and shamelessly biased with my own experience and preferences.

Naming

Naming things properly is harder than it seems, but the effort expended coming up with the best possible identifier name every time, pays tenfold. The ability toname things precisely and correctls develops over time: as a beginner you are less into *abstraction* and tend to name things after what you can see – I need a `Range`, it's called `rng1` and that's good enough. As you progress, you start recognizing the places where `rng1` and `rng2` are no longer sufficiently descriptive and more abstract identifiers like `source` and `destination` start showing up. Procedure names evolve too, from `Macro1` to `CreateInvoice`.There was a time when I'd skip naming form controls that didn't need events handled, or code interactions. Dropping the members list would a bunch of `txtSomething` textboxes, but then also a bunch of `Label42` nameless controls, and the last thing you want in the code-behind is such a meaningless identifier! Name everything that can have a name, whether it's a control on a form or any shape on a worksheet: if it has a name, it has a purpose. If it has a purpose, it should have a name.

Form controls historically have that 3-letter prefix that identifies their control type, a bit like Hungarian notation but for form controls: it's useful to know that we're looking at an input control that has a `Value`, but I will submit that the type of a control doesn't need to be in its name: for example, `txtSomething` becomes `SomethingBox` where `Box` merely implies a user input; the `Something` part should simply be descriptive enough: `StartDateBox` and `PricePointBox` are self-explanatory enough... and their respective event handlers will look like `PascalCase_PascalCase()`, which respects the standard VB casing rules.

Casing

For a long time, I tried using `camelCase` for locals and parameters, and `PascalCase` for modules and their members; VBA being case-insensitive, it doesn't differentiate between `dosomething` and `DoSomething`. If two declarations have the same identifier name, whichever was last written to VBA's internal *names list* "wins" and becomes that identifier's depiction in the source code. The problem is what naming is hard, and then naming things and having to think of something different (like some arbitrary prefix, perhaps) just to avoid constantly re-casing the names list... which is nothing but a distraction at the end of the day, because the language doesn't care about it. That's why I gave up on `camelCase` in VBA, but I wouldn't blame someone for trying. Consistency is much harder to achieve though, and there are many things more important than the casing of the first character of an identifier name...

Purpose

Naming things can be hard sometimes but given that we read any line of code much more often than we write it, taking the time to think about the *purpose* of an abstraction (whether that's a variable, a procedure, a class, or a project) when naming it, is always a good investment.

You have a *summary sheet*, not a "Sheet1". There's an *accept button* and an *instructions label* on that form, not a "Command2" and a "Label8". If you need a comment to explain what's behind any identifier name, it's probably a name that isn't descriptive enough.

That's why you want identifier names that are easy to read and don't require you to make a mental map of what's what, because they're self-explanatory. For the same reason, avoid *changing the meaning* of an identifier at any point during execution: there's plenty of room for every variable you might need; avoid repurposing an identifier not only within a procedure, but throughout your code base.

Avoid declaring a variable for one purpose *and then using it for another*; a `rng` variable to hold some `Range` object reference makes it very easy (if not tempting) to repurpose, versus a more specific name like `InvoiceDateCell`. One identifier, one purpose. If you encounter any given identifier name in your code base, it should mean the same thing everywhere, so you can tell what you're looking at without having to backtrack to where it was last assigned (side note, that's much easier to do for locals vs globals), and working out the meaning of some expression to figure out what the name stands for: the purpose of anything should always be as obvious as possible from its name alone.

Renaming

Naming is hard enough, renaming things should be easy. With Rubberduck's ***Rename*** refactoring (Ctrl+Shift+R) you can safely rename any identifier ***once***, and all references to ***that*** identifier automatically get updated. Without a refactoring tool, renaming a form control can only be done from the ***Properties*** toolwindow (F4), and doing this instantly breaks any event handlers for it; renaming a variable by hand can be tedious, and renaming a single-letter variable like `a` or `i` with a local-scope find/replace

(Ctrl+H) can get funny if the scope has any comments. Rubberduck knows the exact location of every reference to every identifier in your project, so if you have a module with two procedures that each declare a `localThing`, when you rename the local variable `localThing` in the first procedure, you're not going to be affecting the `localThing` in the other procedure. But if you rename `CommandButton1` to `OkButton`, then `CommandButton1_Click()` automatically becomes `OkButton_Click()`.

Never hesitate to rename a bad identifier you come across; be merciless. Naming things after their purpose rather than their nature (we don't care that an amount is a 64-bit `Currency` value, but we do want to be able to tell `Sales` from `Cost` amounts). Does something need more vowels, or a better name crosses your mind as you read the code? Rename it; remove the distraction and let your mind focus on the content rather than its container!

In a nutshell

- Use `PascalCase` if you like. Use `camelCase` if you like. Consistency is what you want to shoot for, and in a case-insensitive language that only stores a single version of any identifier name it's much easier and simpler to just use `PascalCase` everywhere and move on to more interesting things~~, like tabs vs spaces~~.
- Avoid `_` underscores in identifier names, ***especially*** in procedure/member names.
    - Causes *compile errors* with `Implements`.
- Use meaningful names that can be pronounced.
- Avoid disemvoweling (arbitrarily stripping vowels) and *Systems Hungarian* prefixing schemes.
- A series of variables with a numeric suffix is a missed opportunity to use an array.
- A good identifier name is descriptive enough that it doesn't need an explainer comment.
- Use a descriptive name that begins with a verb for `Sub` and `Function` procedures.
    - Note: if you use *explicit* `Call` *statements*, you might be tempted use a descriptive *noun* instead (not `Macro1`!), since "call" becomes the verb/action; consider going with `PrepareWeeklySalesReport` over `WeeklySalesReport`.
- Use a descriptive name (a noun) for `Property` procedures and modules.
- For object properties, consider naming them after the object type they're returning, like `Excel.Worksheet.Range` returns a `Range` object, or like `ADODB.Recordset.Fields` returns a `Fields` object.
- Appropriately name everything the code must interact with: if a rounded rectangle shape is attached to a `DoSomething` macro, the default "Rounded Rectangle 1" name should be changed to "DoSomethingButton" or something that tells us about its purpose. This includes all controls on a `UserForm` designer, too. `CommandButton12` is useless; `SearchButton` is much better. Consider also naming the controls that ***don't*** necessarily interact with code, too: future code might, and the author of that future code will appreciate that the bottom panel is named `BottomPanel` and not `Label34`.
- Never hesitate to rename any identifier you find a better, more descriptive name for.

Parameters & Arguments

When you start breaking down large procedures into smaller ones, you quickly discover a need to pass state and data between procedures. Bumping a local declaration up to module level is one way to do this: all procedures in a module have access to all module-level declarations in that module.

But it's not because two or more procedures in a module need a particular value, that it makes sense to declare that value at module level: *abstraction levels* should be consistent, and declarations within a module should be cohesive and closely related. Therefore, it often makes much more sense to keep things as tightly scoped as possible, and to *pass things around as parameters* instead.

A parameter is a local declaration that is part of the *signature* of a procedure (whether that's a `Sub`, `Function`, or `Property` procedure); the run-time value of a parameter is passed to the procedure via an *argument* expression, wherever the procedure is being called. `Property Let` and `Property Set` procedures need to have an extra parameter that represents the value (or reference) being assigned to the property; this parameter might be best named `Value`, `NewValue`, or `RHS` (it's the *Right-Hand-Side* of the assignment operation).

There's a common source of confusion around the use of parentheses when calling a procedure. To understand when parentheses are required, we need to understand the syntax of a procedure call. There are two reasons we might want to call a procedure: the first is to execute a series of side-effecting statements that either complete successfully or raise an erro.. The second is to retrieve a value from a function or property (normally not a side-effecting call).

Unless we're interested in a return value, parentheses should not be supplied:

```
ProcedureName [ArgumentsList]
```

Where `ArgumentsList` is an optional, comma-separated list of argument expressions that get evaluated just before their value is placed on the stack for the invoked procedure to retrieve.

An argument expression could be a literal value, a variable, a function call, an object, an object's property, or a complex equation involving multiple values and functions. When you write `MsgBox "Hello"`, the string literal value `"Hello"` is passed to the `MsgBox` function's `Message` parameter. If you do `MsgBox Msg`, the variable `Msg` gets evaluated into a value; that value is passed to the function. If you write `MsgBox "Hello " & Name`, the value held by the `Name` variable gets concatenated with the string literal value `"Hello "`, and the *resulting string* is what the function receives.

You could have parentheses around each argument expression in a list of arguments, but the list itself does not have parentheses. The VBE subtly hints at this by consistently forcing a space to appear between the name of the procedure and the first argument expression:

```
MsgBox ("Hello " & Name), ("Title")
```

Treating these parentheses as part of the argument list results in a compilation error::

```
MsgBox ("Hello " & Name, "Title")
```

...Because the parenthesized expression makes no sense and cannot be evaluated.

But what if we *want* to capture a returned value? In that case the parentheses around the argument list are required, and the VBE also hints at it by consistently sticking the opening parenthesis to the procedure's name:

```
PromptResult = MsgBox(Msg, Title, vbYesNo)
```

The `MsgBox` function returns a `VbMsgBoxResult` value that we can discard without a second thought when all we want is to display a message, but we can capture that value in a local variable and use it later if we need to; the parentheses are then required and surround the argument list.

Named Arguments

The expressions supplied in a list of arguments can be named. This is useful in cases where the results of multiple expressions are being passed to a procedure (that's a problem: arguments should be kept simple to facilitate debugging; consider extracting these expressions into local variables, and passing the named variables instead of naming the argument expression), or when passing literals that don't necessarily have an obvious meaning – for example when passing a `True` or `False` Boolean literal to a procedure, naming the argument can help clarify what the Boolean value stands for without needing to bring up *IntelliSense*, peeking at the procedure's definition, or outright navigating to it.

Naming arguments is always optional, and as it can make call sites noisy it should be used judiciously, perhaps in combination with line continuations as appropriate. When you use named arguments with line continuations, make sure you keep the name together on the same line as the associated argument expression: the name and the `:=` operator are both part of the argument and should never be spanning multiple lines.

```
DoSomething Arg1:=”test”, Arg2:=42, Arg3 _
>    :=False
```

One place where named arguments really shine, is when we're passing a variable *by reference* to a procedure that then assigns this parameter to *return* one or more values to the caller. In such cases it's nice to use an "out" prefix to signal that the parameter is an *output* rather than an *input*; the language doesn't have an `out` keyword that could enforce this at the compiler level, so it's up to us to adopt a convention, and use it consistently. `outResult:=SomeVariable` makes it very clear that `SomeVariable` is going to be assigned by whatever procedure we're calling, without needing to peek into the procedure's implementation to find whether and where a `ByRef` parameter might be assigned.

ByRef, ByVal

The (unfortunate) default in VBA is to pass parameters *by reference*. In most cases it won't make a difference but ignoring it can cause subtle bugs and make your life harder than it needs to be. Passing a variable by reference, means the procedure being called is receiving a reference to that variable, rather than just its value. The implication of this, is that the procedure could re-assign that reference and, if the calling code wasn't expecting this, local state can become unpredictable or difficult to debug.

When you pass a parenthesized expression to a procedure that accepts it by reference, VBA evaluates the expression and passes its result to the procedure; in other words, you cansurrounde an argument with parenthese when the parameter is passed `ByRef` but you do not wish to pass a reference to, say, a variables:

```
DoSomething (SomeVariable)
```

Here `DoSomething` is receiving a reference to the result of the expression that evaluates the value of `SomeVariable`; without the parentheses, the procedure would be receiving a reference to `SomeVariable`, and if the procedure scope re-assigns the value held by the corresponding `SomeParameter` parameter, because the parameter is passed `ByRef` the address of `SomeParameter` is the same as that of `SomeVariable`, so when it gets re-assigned in `DoSomethin`g, it's re-assigned everywhere else that reference was being held, too.

Because a procedure shouldn't normally be allowed to interfere with the state of its caller, we will prefer consistently passing parameters `ByVal` whenever possible – especially when passing object references. `ByRef` is useful when a procedure obviously and explicitly must receive a parameter by reference, for example when you write a `TrySomething` function that returns a `Boolean`, and then conditionally affects an *output parameter*. Some data types cannot be passed by value: arrays and user-defined types should always be passed by reference unless they're being wrapped in a `Variant`.

In a nutshell

- Prefer passing values as parameters instead of bumping the scope of a variable to module-level, or instead of declaring global variables.
- Pass parameters `ByVal` whenever possible.
    - Arrays and User-Defined Type structures cannot and should not be passed by value.
    - **Objects are never passed anywhere** no matter the modifier: it's only ever (`ByVal`: a copy of) a ***pointer*** that gets passed around – and most of the time the ***intent*** of the author is to pass that pointer ***by value***. A pointer is simply a 32-bit or 64-bit integer value, depending on the bitness of the process; passing that pointer `ByRef` (explicitly or not) leaves more opportunities for programming error..
- Use an explicit `ByRef` modifier whenever passing parameters by reference.
- Consider specifying an `out` prefix to name `ByRef` return parameters.
    - Consider using named arguments for `out`-prefixed `ByRef` return parameters.

- Use parentheses when capturing a procedure's return value, either into a local variable or when directly passing the result to another procedure as an argument.
- Discard parentheses around arguments when discarding a procedure's returned value, or when calling a `Sub` procedure (which doesn't return anything).

Comments

Good comments should be about *why* something is happening, *why* something is being done a certain way. Bad comments are always redundant distractions that should be removed: let the code explain the *what* and the *how*. Note that I make a distinction between comments in the code, and documentation: in VBA every module and member can have a hidden `VB_Description` attribute that can be leveraged to document the purpose of the module or member with that attribute. With Rubberduck you can maintain these hidden attributes and surface them as comment annotations; these are a different type of comment that are meant to be treated as developer documentation – *docstrings* that appear in the IDE's *Object Browser* tool, and exposed in several additional places in the Rubberduck UI (Code Explorer, contextual toolbar, and eventually in custom *IntelliSense* tooltips as well). These specific comments are usually fine being about the *what*, since when you read them, you want to be sure you're calling the right thing.

This isn't about such documentation, rather everything else: comments next to a variable declaration, comments between two chunks of code in the middle of a procedure; comments that distract from the code or otherwise don't belong. ASCII art has its place in a console application's splash/startup, but not in the middle of a code base as a huge annoying comment. Copyright, license information belongs in a README and/or LICENSE file (or since this is VBA, this metadata could live somewhere in the host document), not as a 40-liner comment header at the top of every module. Versioning, change sets, support ticket numbers: none of it belongs in comments, use the appropriate version control tooling instead (git comes to mind).

Cleaning up old comments in a legacy or inherited project can be difficult: you read the comment, then the code it's for, then you read the comment again, and it takes you a few precious seconds to realize you're being played – the comment says one thing, but the code says another. This is why we want comments that say *why* much more than comments that say *what*. You want a comment to be there to save the day when it reminds you that a seemingly useless variable exists to more easily break at a strategic place when debugging, or a comment that tell its reader that we need to lookup the *average* cost associated to a SKU in order to calculate the correct margins, so that's why the lookup is against the inventory table and not against the purchases table. Good comments are invaluable knowledge that you cannot infer from the code alone: let the code speak for itself and be the only source of truth about what it's doing.

There aren't too many ways to write non-distracting comments in VBA, but it might puzzle more than a single future maintainer to denote comment with the legacy `Rem` marker/keyword rather than the much more common single quote character. Line continuations should be avoided as well: there's no `/* comment block */` syntax in VBA, and new lines delimit instructions (rather than an explicit terminator like a

semicolon); a multiline comment with line continuations is unnecessarily harder to maintain and keep together than a series of consecutive comment lines would ever be. Because VBA will parse a line-continued comment as a single *logical line of code*, comments should never end with an underscore preceded by a space: doing this makes the next line (usually a declaration) be a continuation of the comment, not an instruction.

In a nutshell

- Use the single quote `'` character to denote a comment.
- Avoid line-continuing comments; use single quotes at position 1 of each line instead.
- Consider having a `@ModuleDescription` annotation at the top of each module.
- Consider having a `@Description` annotation for each `Public` member of a module.
- Remove comments that describe ***what*** an instruction does, replace with comments that explain ***why*** an instruction needs to do what it does.
- Remove comments that summarize ***what a block of code*** does; replace with a call to a new procedure with a nice descriptive name.
- Avoid cluttering a module with banner comments that state the obvious. We know they're variables, or properties, or public methods: no need for a huge green comment banner to tell us.
- Avoid cluttering a procedure scope with banner comments that split up the different responsibilities of a procedure: the procedure is doing too many things, split it up ***and appropriately name the new procedure*** instead.

Variables

When you consider that any given instruction could possibly go wrong and blow up in your face, you tread carefully and handle run-time error.. And then what? If you're chaining member calls and supplying function results as arguments, and these functions are themselves receiving function results for their own parameters, ...you may have been able to accomplish everything you need in a one-liner, but the cost is that if anything goes wrong anywhere, debugging will not be a pleasant nor efficient exercise. Using variables to hold intermediate values or references allows you to stop and validate things before you proceed to consume them. And when debugging, it makes it easier to review the outcome of each individual operation. Often, we can avoid raising a run-time error by capturing a reference or a value into a variable, and then bailing out instead of continuing with invalid state or data.

Variables are a form of abstraction: a variable might represent the result of a function, or that of a complex equation that has a name. One nice thing about variables is that they not only have a meaningful name,but they also have a *data type*. If you omit the data type, you get a Variant; it's better to be explicit about data types, mindful about them, *because* the language often doesn't mind them at all, and that can spell trouble when you're not familiar with all the other implicit calls and conversions VBA can do at run-time: declaring an explicit data type not only signals your intent, it also enables early binding: member calls can be validated at

compile-time when they're early-bound. Otherwise, even `Option Explicit` cannot save you from a typo in a late-bound member call. Using variables keeps things earl -bound and (hopefully well) named.

At the module level, keep declarations to a minimum, ideally of `Private` encapsulated state and `WithEvents` object references (mostly in class modules then). The implications of both `Private` and `Public` fields in a standard module can be hard to reason about (think about what the `End` keyword does to that state), and they aren't necessarily innocuous. The good news is that corrupt global state doesn't really happen when you don't rely on global state! That's part of the advantages of keeping scopes tight and passing parameters around: state rarely ever really *needs* to be global.

Avoid duplicating in local scope what's already global. For example, class types with a `VB_PredeclaredId` attribute set to `True`, make VBA define a project-wide object reference to a *default instance* of the class; that *default instance* is always named after the class itself, and to always be crystal-clear about our intent, we should use *that* identifier to refer to *that specific* instance of the class.

For example, in Excel, `ThisWorkbook` always represents the host document – the Macro-enabled Excel workbook file that contains the VBA code we're looking at; there's only ever one `Workbook` object in the host document, and *within that document* any VBA code that refers to that workbook object should do so using the `ThisWorkbook` identifier. Storing that reference in a local variable with a different name serves no purpose and adds a layer of *indirection* that obscures what's really going on and duplicates the already-global object reference.

Variables should be declared in a procedure scope *as their presence becomes relevant*. This might be a major clash with your coding style, but imagine if the introduction to *Lord of the Rings* or that of *Game of Thrones* told you the names and family tree of every single character that's ever going to be seen in each movie or episode, and then that information would otherwise disappear from the script: you would quickly lose the plot and constantly be wondering who's who anyway. Instead, that knowledge is gradually introduced to the viewer and they can focus on the plot unfolding as they absorb the new information.

Saturating the reader's thoughts with any amount of not-relevant-yet (if at all!) information by declaring locals as a literal *block* of declarations at the top of a scope, has the same effect: *bombarding* someone with information usually just isn't considered a sensible approach. In my experience, declaring locals at the top puts the code at a much higher risk to leave a sometime- surprising number of unused declarations in place.

Declaring locals as you need them and require their assignment, puts the declaration in its usage context, which could help reduce the need for explanatory comments, and addresses every single issue pointed out above. This clearly isn't a "because everybody else does it" type of argument (even though most other languages *do* typically declare things as they are needed).

This also implies declaring locals *separately*, with one `Dim` statement per variable, followed by the assignment of that variable. I like ty keep these statement pairs together and with breathing space both above and below: if a procedure is doing one thing and doing it well, often it just naturally doesn't need too many variables anyway.

In a nutshell

- Declare all variables, always. `Option Explicit` should always be enabled.
- Declare an explicit data type, always. If you mean `As Variant`, make it say `As Variant`.
- Consider using a `Variant` to pass arrays between scopes, instead of typed arrays (e.g. `String()`).
    - Pluralize these identifier names: it signals a plurality of elements/items much more elegantly than *Pirate Notation* (`arr*`) does.
- Avoid `Public` fields in class modules; encapsulate them with a `Property` instead.
- Consider using a backing user-defined `Private Type` structure for the backing fields of class properties; doing so eliminates the need for a prefixing scheme, lets a property be named exactly as per its corresponding backing field, and cleans up the *locals* toolwindow by grouping the fields under a single module variable.
- Limit the scope of variables as much as possible. Prefer passing parameters and keeping the value in local scope over promoting the variable to a larger scope.
- Declare variables where you're using them, ***as you need them***. You should never need to scroll anywhere to see the declaration of a variable you're looking at.

Late Binding

Declaring and using variables with a specific data type, more specifically *object types*, enables consistent *early binding*, where both the compiler and static code analysis tools have the best possible understanding of the code. Late binding (the opposite: binding types and members is deferred to run-time) has little to do with `CreateObject` and whether a library is referenced. In fact, *late binding* happens implicitly rather easily, and way more often than would be healthy. Strive to remain in early-bound realm all the time: when the compiler / *IntelliSense* doesn't know what you're doing, you're on your own, and even `Option Explicit` can't save you from a typo (and error 43)..

You know you're making a late-bound member call when you type the *member access* operator (dot) and nothing happens when earl -bound, this normally triggers *IntelliSense* and a completion list to appear in the editor. Without the compile-time type information, the editor cannot assist you and the compiler will accept whatever you type as a member call, deferring its resolution to run-time. When this happens, you want to stop what you're doing, figure out what interface you intend to be working with (make sure the run-time object actually implements that interface, otherwise you'll get a *type mismatch* run-time error), declare a local variable with that type, assign it to the expression you were about to chain a member call to, and then make your early-bound member call.

For example, when you retrieve a `Worksheet` object from a `Sheets` collection in the Excel object model, what you get is an `Object` that may be a `Worksheet`, a `Chart`, or about half a dozen more legacy sheet types; most of the time what you get is indeed a `Worksheet`, but the object model cannot assume this, and neither should your code. Any member call chained to this retrieval will be late-bound and can fail with error

438 if our assumptions are erroneous. That's why we prefer using the `Workbook.Worksheets` collection over the `Workbook.Sheets` collection to retrieve a `Worksheet` instance: both return `Object` and need to be cast into a `Worksheet`, but only one of them guarantees we're getting a `Worksheet` object (or another `Sheets` collection if we gave it an array of sheet names).

The *dictionary access aka "bang"* operator (`!`) is also a feature that seems to have been added to facilitate *writing* code with little to no consideration for *reading* that code. And since the `!` token is syntactically ambiguous (it's also the *type hint* character for a `Single`), it can considerably slow down parsing because the ambiguous grammar takes more iterations to resolve the correct parser rule.

The operator works by querying the object for a *default member* that takes a single `String` parameter; the "identifier" that follows the operator is expectedly (?) the `String` argument that is passed to the *default* member, hence the common need for square brackets when the names involved are allowed to include spaces. But that's not the whole story: in many cases, what the *default member* returns is a reference to another object that *itself* has a default member, and *that* is what the *dictionary access* expression resolves to: something that is a complete black box in terms of the class types involved: this operator performs implicit voodoo magic that puts a lot of wheels in motion just to avoid spelling out `Recordset.Fields("FieldName").Value` when that could easily become a `GetFieldValue(Recordset, "FieldName")` function call that spells it out *once* and *comments-out* the equivalent `Recordset!FieldName` incantation.

In a nutshell

- Avoid making a member call against `Object` or `Variant`. If a compile-time type exists that's usable with that object, a local variable of that data type should be assigned (`Set`) the `Object` reference and the member call made early-bound against this local variable.
    - Taking an object presenting one interface and assigning it to another data type is called "casting" If the object doesn't support/implement the new interface, the cast/conversion fails..
- Of course, ***explicit*** late binding is OK (`As Object`, and create instances of classes from unreferenced libraries with `CreateObject` instead of the `New` operator). Late binding is very useful and has many legitimate uses, but generally not when the object type is accessible at compile-time through a library reference.

Avoid the ***dictionary-access*** (aka "bang") operator `!`, it is late-bound by definition, and makes what's actually a string literal read like a member name, and any member call chained to it is inevitably late-bound too. Rubberduck can parse and resolve these, but they're much harder to process than standard method calls.

Explicitness

Rubberduck encourages you to be consistently explicit about certain things, and consistent implicit about others. For example, the optional identifier name after a `Next` keyword is considered redundant, because a loop body that only contains a single parameterized procedure call cannot span so many lines you lose track of what you're iterating! However, an explicit `ByRef` or `Public` modifier, even if redundant, signal a much clearer intent than their absence does.

But *explicitness* goes beyond optional tokens in the language syntax: it's about avoiding implicit references, for example when you're automating in Excel but you use `Word.Documents` instead of retrieving that document collection from a specific `Word.Application` object reference: this inevitably leads to unexpected outcomes.

Many libraries expose classes with a parameterless *default member;* when you mean to refer to that member, do it explicitly. For example, `Application.Name` is implicit in `Debug.Print Application`, and it shouldn't be, because there is no way to read the implicit member call from the code alone: the reader *must know* that this particular *application* object class will let-coalesce to its *name* property. The only way to know is by looking it up in the *object browser*, where hidden members can be revealed; a class's' default member will have a small blue dot overlay on top of its icon.

In a nutshell

- Use explicit modifiers everywhere (`Public`/`Private`, `ByRef`/`ByVal`).
- Declare an explicit data type, even (especially!) if it's `Variant`.
- Avoid *implicit qualifiers* for all member calls: in Excel watch for *implicit ActiveSheet references*, *implicit ActiveWorkbook references*, *implicit containing worksheet references*, and *implicit containing workbook references*, as they are an extremely frequent source of bugs.
- Invoke parameterless *default members* explicitly.
    - Note: some object models define a *hidden* default member (e.g. `Range.[_Default]`) that redirects to another member depending on its parameterization. In such cases it's best to invoke that member directly; for example use `Range.Value` as appropriate, but the hidden `[_Default]` member is better off not being invoked at all, for both readability and performance reasons.
- Invoke parameterized default members *implicitly* when they are indexers that get a particular item in an object collection, for example the `Item` property of a `Collection`. Invoking them *explicitly* doesn't hurt but could be considered rather verbose.
- `Call` is not a keyword that needs to be in your program's vocabulary when you use expressive, descriptive procedure names that *imply* an action taking place.
- Consider explicitly qualifying standard module member calls with the project (and module) name, including for standard and referenced libraries, especially in VBA projects that reference multiple object models.

Error Handling

Should every procedure systematically have an error-handling subroutine? For me the answer is a solid "no" because I like to control what errors *bubble out of* methods at run-time, and having an error-handling subroutine that simply forwards the error to the caller, is redundant.

To understand error handling in VBA, you need to first understand how VBA behaves at run-time.

Happy Path

When VBA enters a procedure scope, there's only a "happy path". An `On Error` statement instructs VBA to jump to a local error handling subroutine instead of jumping out of the procedure scope entirely.

```
Public Sub DoSomething()
    On Error GoTo CleanFail
    ' ...happy path here
CleanExit:
    ' ...cleanup code here
    Exit Sub
CleanFail:
    ' ...error path here
    Resume CleanExit ' comment-out to debug
    Stop
    Resume
End Sub
```

It's important to cleanly separate the code that is allowed to run in a run-time error recovery state from the code that is merrily going about doing its business – the "happy path" should never be reachable while an error is being handled.

Intertwining happy and error paths will make you wonder what's going on when a runtime error occurs despite the `On Error` statement.

Error Path

Given an active `On Error` statement, when an error occurs in the "happy path", we are taken into the error handling subroutine, and VBA enters an error recovery state: the `Err` object contains information about the runtime error, `Resume` can jump us right back at the statement that raised the error.

What you want to do here, is recover the state if possible and resume normal execution. If you cannot recover from the current state, use a `Resume` instruction to clearly reset the error state and jump to a label at the end of the happy path that runs whether an error occurs or not – that way execution always exits from the same place, and you know something is wrong when it doesn't. Avoid raising errors while in an error state, unless you mean to "rethrow" an error up the call stack for the caller to handle:

By re-raising the same error number, we don't lose the description in the global `Err` object:

```
Public Sub DoSomething()
    On Error GoTo CleanFail
    ' ...happy path here
CleanExit:
    ' ...cleanup code here
    Exit Sub
CleanFail:
    ' ...error path here
    With Err
        Select Case.Number
            Case Errors.ERR_SomeCustomError
                ' ...
>               Resume CleanExit
            Case Else
                .Raise .Number ' rethrow
        End Select
    End With
    Stop
    Resume
End Sub
```

What's critical is that at no point during the execution of an error-handling subroutine does execution ever jump back into the *happy path* without clearing the error state. The safest way to do this, is to ensure the error path always encounters a `Resume` statement.

The `CleanExit` label might only stand for an `Exit` statement, but its role is literally to just exit the procedure *cleanly*, without entering the section that's meant to run while we're in an error state. For example, if the scope owns a database connection, that's a good spot to clean it up.

Structured Programming (Procedural)

Before structured procedural programming, we didn't really have *modules* in BASIC; you would write a script and carefully *number every line* to make sure you have enough of a gap in the numbering (typically you'd jump 10 at every line) for your "module" to grow, and then `GOTO` and `GOSUB`/`RETURN` statements would take you to and from *subroutines* within the script. Procedural programming was pretty much a whole new paradigm when it appeared: now we would be able to organize a large project into specialized module, and organize these modules within themselves with actual named procedure scopes instead of leaving gaps in the line numbering. With *structured error handling* coming with the `Err` object, line numbering is in fact completely archaic and useless in modern code.

Much of the age-old BASIC code still works (with minor adjustments) in VBA: until .NET, the language evolved without breaking much in terms of backward compatibility. As a result, there are plenty of *paleo-constructs* and keywords that exist in the language to preserve this compatibility; their existence does not mean we should be using them: there's a reason they were superseded by other keywords and constructs. `While…Wend` comes to mind: there's no rational reason to use it over a much more standard and flexible `Do While…Loop` construct. `Error$` statements give you the current error message string, but in modern code you want to use the `Err` object instead. `Global` really means nothing more than `Public` in VBA. Nobody writes `On Local Error` anymore, and yet it and everything else is supported and will work as expected.

Still, *procedural programming* is a perfectly well adapted paradigm to write VBA code in: you'll be consuming objects (perhaps unknowingly, sometimes!) more than you'll be writing your own, and that's fine! What matters is that you keep things *structured*, not just *procedural*. Keep separate macros in separate modules, with a very high-abstraction `Public` procedure at the top that calls lower-abstraction `Private` procedures with increasingly more excruciating details about what's going on: having consistent abstraction levels is an art: you want to be working with cells and ranges and Win32 API at the lowest level of abstraction; the higher levels put a name on what's happening, so you read it as "prepare the report heading" instead of writing a number of values in a number of cells.

The age-old programming principles all apply with procedural programming: *Don't Repeat Yourself* (DRY) and *Keep It Simple, Stupid* (KISS), notably. So you will want to structure your modules like stories that unfold as you dig into their lower-abstraction private procedures: this implies that your typical module only contains a single public procedure at the top, embodying a macro at its highest abstraction level.

Inevitably, you'll encounter a situation where you write another macro/module, and realize that much of the lower-abstraction stuff is very similar – and that's when you copy & paste a chunk of code into a new module? No! That's when you extract a procedure into its own module with `Option Private Module`, make the extracted procedure `Public`, and invoke it from both places.

Over time you'll find that the helper procedures that worthy of such a treatment are usually very simple, common operations that benefit from being named – for example if you're calculating a gross margin in multiple places, it might be useful to extract a function that takes a cost and a selling price as parameters, validates the inputs, and returns the result given a non-zero selling price (and -1 in such a case).

Validating inputs should be systematic: we never want to divide anything by zero, so instead of making it the callers' responsibility, a function should check that the operation can be executed safely *before* causing an avoidable run-time erro.. Sometimes invalid inputs can be dealt with directly: here the `CalculateGrossMargin` function knows it cannot calculate a result without a price amount, but then it also makes sense to return -100% when we're giving away something that cost a non-zero amount. On the other hand, a *negative* cost amount wouldn't make sense and the function should refuse to compute this input; raising a custom run-time error with a helpful message is the best thing to do then, because this would be an *exceptional* problem on the caller's side.

Being *defensive* about your inputs, as in not trusting any of them to contain the expected values (or range of values), makes your code much more robust, and when things go wrong you know right away. Imagine we somehow got an invalid negative cost amount in our data, and then the gross margin percentage is used somewhere else to compute a forecast of some sort: without input validations, you would pick up the problem at the end of the process, when the report comes up with erroneous figures. *With* input validation, the macro can stop as soon as it encounters the bad data, and the validation error can be shown to the user.

Other times the data comes directly from the user: when dealing with user inputs we have to be even more defensive, and code as if the user was doing everything in their power to break our code. Maybe a string is empty and expected not to be; maybe a currency code doesn't exist in your lookup table; maybe there was a typo and they entered `456` instead of `123`, or a date is set in the future and shouldn't be, or maybe they used a different decimal separator when they entered an amount, and your code cannot assume all numbers will look the same for all users. Or a prompt could just be cancelled; there are so many ways for everything to go wrong with user inputs, and not validating them, or assuming they're as expected, is inevitably going to make as many bugs surface… and never at an appropriate moment.

You will want your top-level procedure to handle any errors that may be unhandled by the lower-level ones, but you will want to treat *programming errors* differently than *user error*:: it's fine for a programming error to halt code execution with a cryptic (native) error message, but typically you will want to have much clearer/friendlier messages shown to your users when the problem is up to the user to fix. For example "SKU 1234 has no selling price", instead of a native "division by zero" error that occurs somewhere during execution, well after validation would have picked it up.

Low-abstraction code is where unexpected errors are more likely to occur, because such code is almost always at the boundary of another system: something as trivial as reading a worksheet cell value into a local variable can go wrong in multiple ways if you make assumptions (trust worksheet cells as much as user inputs – that is, *not at all*!). Working with files, a file system, a network, any type of connection; all these things depend on components your code has no control over, that should be able to fail gracefully.

In a nutshell

- One macro/script per module. Do have it in a module rather than a worksheet's code-behind.
- `Public` procedure first, followed by parameterized `Private` procedures, in decreasing abstraction level order such that the top reads like a summary and the bottom like boring, small but specific operations.
    - You know it's done right when you introduce a second macro/module and get to pull the small, low-abstraction, specific operations into `Public` members of a utility module, and reuse them.
- Don't Repeat Yourself (DRY).
- Consider passing the relevant state to another procedure when entering a block of code. Code is simpler and easier to follow when the body of a loop or a conditional block is pulled into its own scope.
- Avoid using error handling to control the flow of execution: the best error handling is no error handling at all, because assumptions are checked and things are validated. For example instead of opening a file from a parameter value, first verify that the file exists instead of handling a ***file not found*** error... but still handle errors, for any ***exceptional*** situations that might occur while accessing the file.
- When it's not possible to avoid error handling, consider extracting a `Boolean` function that swallows the expected error and returns `False` on failure, to simplify the logic.
- Handle errors around all file and network I/O.
- Never trust user inputs to be valid or formatted as expected.

Object Oriented Programming

In VBA/VB6 we get to go further than mere scripting and apply Object-Oriented Programming (OOP) principles, probably more relevantly so in VB6 and larger VBA projects. For many years it has been drilled into our heads that VBA/VB6 cannot do "real" OOP because it doesn't support inheritance. The truth is that there is much, ***much*** more to OOP than inheritance, and ***if you want to learn*** and apply OOP principles in your VBA/VB6 code, you absolutely ***can***, and you absolutely ***should***, and Rubberduck will absolutely help you do that.

The principles that apply to procedural programming are also valid for OOP, but OOP involves mostly classes and objects. There are two main types of objects: data, and services.

Data classes are meant to encapsulate data; in Java they're called *beans*, C# calls them POCO (Plain Old CLR Objects), or you might have seen the term DTO (Data Transfer Object) before; they're all the same thing – objects that *do* nothing but encapsulate data: these objects *are* the data and they usually look like a simple module that exposes a bunch of properties.

The other type of class encapsulates a process; a *stateful* one may encapsulate its own state/data, and might even expose properties for this, but it also has *methods* that *do something* with the data. Such a service class may depend on other services, that themselves have their own dependencies.

In OOP when a class creates an instance of another class, or when it consumes a particular specific "concrete" class, it is said to be *coupled*. Coupling isn't inherently evil: you generally *want* tightl -related classes to be working together. But when you need to *decouple* a class from another, OOP gives you more tools to achieve better abstractions: you can extract an *interface* from a concrete class, and then this interface becomes an abstraction you can work with, because the code that consumes an abstract interface should never need to care about the specific concrete run-time type of an object.

At some point you *do* need to know what specific implementations you're going to be working with; using *dependency injection* (DI) we can pick an implementation at start-up, and inside (or near) the *entry point* procedure we create new instances of each object we need to inject.

Dependency injection works nicely with *composition*, which happens when an object encapsulates another; classes can have "has-a" relationships in VBA, but not "is-a"; class inheritance isn't supported. Classes in VBA don't have constructors, so how can we inject our dependencies?

While *constructor injection* is the preferred way to do this in other languages, every other DI technique can be implemented and used in VBA, starting with *property injection*, combined with *factory methods* that leverage class modules' *default instance*. If you are not familiar with the DI terminology, *dependency injection* is just a fancy way of saying "passing parameters around". A "dependency" is an object that your service class needs in order to be able to do its thing; for example, a service class that exposes methods to save something in a database would depend on ADODB library types (`Connection`, `Command`, `Recordset`); a class that depends on this service class would therefore be *coupled* with ADODB if it depended directly on it. To decouple ADODB from the rest of the code, we make our database-connected service implement an abstract interface, and the code that needs to use the ADODB service can now depend on this abstraction instead; the initialization code is responsible for creating the "concrete" ADODB service and inject that instance into our service class.

What gives? All that just to end up connecting to a database anyway? The difference is that it's now very easy to make your service class save to a text file instead, by injecting another implementation of the abstract interface. Code (tests) could now be written to verify and prove that the service class is doing its job, *without needing to hit a database*. If your data access interface involves deleting records, being able to test every functionality of your application without connecting to a database has its advantages.

Using Dependency Injection makes it easier to adhere to the language-agnostic SOLID principles at the core of object-oriented programming: separating a class from its dependencies helps following *single responsibility*. Having an abstract interface to depend on satisfies *dependency inversion*, makes it easier to follow *open/closed*, and if you avoid special-casing any particular implementation of a given interface (i.e. a service class is never allowed to know anything about any specific implementation of any of its dependencies) then you've got *Liskov Substitution* covered; keep your interfaces as clear and simple as possible (ideally an abstract interface only has a single member), and you've nailed *Interface Segregation*.

Proper DI occurs at one single place in the code; that place is dubbed the *composition root*, because that's where the application's *dependency graph* gets created. Why a graph? Because if you have a *root* object (say, some App class), and that object has its dependencies, and each of these dependencies can also have their own dependencies, and so on – you get an *object graph*, a bit like the Excel object model.

The composition root should be as close as possible to the entry point of your application, but in VBA the entry point isn't always crystal-clear, nor is it consistently in the same place: if you are writing an add-in, the best place to do this is probably the `Workbook_Open` handler in the `ThisWorkbook` module; the object graph would live at instance level in `ThisWorkbook`, meaning it remains "alive" for as long as the add-in is loaded in Excel. If you're in a macro-enabled workbook, the same might work for your purposes (only now the graph lives as long as the workbook remains open), but the entry point might also be some `Click` handler, or some public procedure/macro that's attached to some shape or button, and there can be multiple entry points then: where the best place is in *your* project depends mostly on whether and how you intend to manage the lifetime of all these objects. The simplest way is to handle `Workbook_Open` and let your app liv fore as long as thehost file is open, and not worry about cleaning anything up; if that causes problems, you may need to handle `BeforeClose` and properly tear down your app.

Sometimes we don't necessarily want to create the app instance at start-up, because our app is just a bunch of not-really-related macros that each do their thing and the user may run none, only one, or any of them and we don't want to spawn objects we're not going to need.

Other times we simply cannot create a dependency at start-up, either because we're missing information that the user must provide at run-time, or we simply don't want a particular instance of a class to live that long. In such cases where we need to defer the creation of a dependency, we can inject an *abstract factory* instead, and only invoke its *Create* method (passing in the required run-time values as arguments) when we're ready to create and consume it. Note that the class that creates an object from a factory, should be responsible for properly destroying that object (i..,. close file handles, connections).

In a nutshell

- Adhere to standard OOP best practices, they are general, language-agnostic concepts that couldn't care less about the capabilities of VBA/VB6:
    - **Single Responsibility Principle** – each abstraction should be responsible for one thing.
    - **Open/Closed Principle** – write code that doesn't need to change unless the purpose of the abstraction itself needs to change.
    - **Liskov Substitution Principle** – code should run the exact same execution paths regardless of the concrete implementation of a given abstraction.
    - **Interface Segregation Principle** – keep interfaces small and specialized, avoid a design that constantly needs new members to be added to an interface.
    - **Dependency Inversion Principle** – depend on abstractions, not concrete implementations.
- Leverage *composition* where *inheritance* would be needed.
- You cannot have *parameterized constructors*, but you still can leverage *property injection* in factory methods to inject *instance-level* dependencies.
- Leverage *method injection* to inject *method-level* dependencies.
    - Avoid `New`-ing dependencies in-place, it *couples* a class with another, which hinders testability; *inject* the dependencies instead.
    - Use the `New` keyword in your *composition root*, as close as possible to an *entry point*.
    - The `Workbook_Open` event handler (Excel) is a possible entry point.
    - Macros (`Sub` procedures invoked from outside the code) are also valid entry points.
    - Let go of the idea that a module must control every last one of its dependencies: let *something else* deal with creating or dereferencing these objects.
- Inject an abstract factory when a dependency cannot or should not be created at the composition root, for example if you needed to connect to a database and wish to keep the connection object as short-lived and tightly scoped as possible.
- Keep the *default instance* of a class *stateless* as much as possible. Actively protect/guard against accidental misuse by throwing a run-time error as necessary.
- Use standard modules instead of a utility class with a `@PredeclaredId` annotation, that never gets explicitly instantiated or used as an actual object.

User Interfaces

UI code is *inherently* object-oriented, and thus a `UserForm` should be treated as the object it wants to be. The responsibilities of a user interface are simple: display and collect data to/from the user, and/or offer a way to execute *commands* (which typically consume or otherwise manipulate the data).

In VBA we can make our own custom user interfaces with a `UserForm` class module, which comes with a `VB_PredeclaredId` attribute set to True – which means that like `ThisWorkbook` and the worksheet modules, we have a globally-scoped identifier (named after the form class) that refers to the *default instance* of the form class. However, workbooks and worksheets are *document modules* that are owned by the host application (Excel), so there isn't any way for these to possibly go wrong; user forms however, are owned by the VBA project and you get to create as many instances as you need for your purposes. This can easily become a problem, because code that refers to the *default instance* will affect *that specific instance*, which may or may not be the instance that's being shown, depending on how you go about using your form. For this reason, we should treat a form's *default instance* the same way we'd treat any other class module's *default instance*: as a *stateless* global object representing the *class type* itself – never to be shown to the user. When we need to display a form, we create a *new* instance of the form class and show *that* instance.

UI code can quickly get messy, because it's very easy to mix responsibilities and have a form be responsible for not only displaying data, but also *acting* on it and even running the entire show. If we implement the functionality directly in buttons' `Click` event handlers, we'll end up with a lot of "business logic" code that has nothing to do with UI and presentation. To keep things tidy and make it easier to eventually refactor a *Smart UI* towards a more structured architecture, the "business logic" should be extracted into its own separate, dedicated module.

Even without adopting a full-fledged object-oriented solution, pulling the form's *data* into a *model* class, and having the form controls manipulate the data in that class instead of directly manipulating a worksheet can make a significant difference in how complex the UI code is to maintain and extend, whether the form is *modal* or not.

A *modal* form is essentially a *dialog* that is shown to the user and blocks execution until the form is closed; it is very much transactional in nature, so offering *cancel* and *confirm* buttons makes sense. If the form is cancelled, nothing happens. If the form is confirmed, a procedure takes the model and consumes its data to perform some operation.

Forms are shown *modally* by default, but you can also display them non-modally. In that case the form is shown, but execution continues immediately after without waiting for the form to be closed. Managing a non-modal form instance that isn't the *default instance* demands a different paradigm altogether, because then user interactions happen asynchronously and we're very much in an *event-driven* paradigm, whereas with a *modal* form there's a clear "before it's shown" and "after it has closed" distinction that makes it easier to reason about.

QueryClose

When making a modal form, it's important to always handle the `QueryClose` event to prevent the form instance from self-destructing and causing issues: the [X] button in the form's control box will destroy the object, which is a already problem if you're not using a *model* and query the form's controls after it has been closed... it's just that as a rule of thumb, you don't expect an object instance you just spawned into existence to spontaneously self-destruct and become an invalid reference; the code that creates an object should be the code that's responsible for the lifetime of that object, and a self-destructing form instance challenges that basic assumption.

That's why you want to make it so that the only way a form ever closes is by merely hiding itself, which resumes execution in a safe manner regardless of how the form instance was spawned: if it's the default instance, you're not resetting its state. If it's a new instance, it's not getting wiped out of existence and the client code (the caller) can safely consume its state if it wants to.

```
Private Sub UserForm_QueryClose(ByRef Cancel As Integer, ByRef CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        Me.Hide
    End If
End Sub
```

Caveat: Microsoft Access Data-Bound UI

VBA projects hosted in Microsoft Access can absolutely use `UserForm` modules too, but without Rubberduck you need to hunt down the IDE command for it because it's hidden. Instead, in Access you mostly create *Access Forms*, which (being *document* modules owned by the host application) have much more in common with a `Worksheet` module in Excel than with a `UserForm`.

The paradigm is different in an Access form, because of *data bindings*: a data-bound form is inherently coupled with the underlying database storage, and any effort to decouple the UI from the database is working directly *against* everything Access is trying to make easier for you.

Treating an Access form the way one would treat a worksheet UI in Excel puts you in a bit of a different mindset. Imagine the Battleship worksheet UI implemented as an Access form: the game would be updating game state records in the underlying database, and instead of having code to pull the game state into the UI there would only need to be code to *re-query* the game state, and the data bindings would take care of updating the actual UI – and then the game could easily become multi-player, with two clients connecting to the database and sharing the same game state.

This is very fundamentally different than how one would go about getting the data into the controls without such data bindings. Binding the UI directly to a data source is perfectly fine when that data source happens to be running in the very same process your VBA code is hosted in: Access' *Rapid Application Development* (RAD) approach is perfectly valid in this context, and its global objects and global state make a nice beginner-friendly API to accomplish quite a lot, even with only a minimal understanding of the programming language (and probably a bit of Access-SQL).

If we're talking about *unbound* MS-Access forms, then it's probably worth exploring *Model-View-Presenter* and *Model-View-Controller* architectures regardless: in such exploratory OOP scenarios the above recommendations can all hold.

In a nutshell

- Avoid working directly with the form's *default instance*. `New` it up instead.
- Form module / code-behind should be strictly concerned with *presentation* concerns.
    - Do implement UI logic in form's code-behind, e.g. enable this control when this command says it can be executed, or show this label when the model isn't valid, etc.
- Consider creating a *model* class to encapsulate the form's state/data.
    - Expose a read/write property for each editable field on the form.
    - Expose a read-only property for data needed by the controls (e.g., the items of a `ListBox`).
    - Controls' `Change` handlers manipulate the *model* properties.
    - Expose additional methods and properties as needed for data/input validation.
    - Consider having an `IsValid` property that returns `True` when all required values are supplied and valid, `False` otherwise; use this property to enable or disable the form's `Accept` button.
- Avoid implementing any kind of side-effecting logic in a `CommandButton`'s `Click` handler. A `CommandButton` should *invoke a command*, right?
    - In procedural code the command might be a `Public Sub` procedure in a standard module named after the form, e.g., a `SomeDialogCommands` module for a `SomeDialog` form.
    - In OOP the command might be a property-injected instance of a `DoSomethingCommand` class; the `Click` handler invokes the command's `Execute` method and could pass the *model* as a parameter.
- Consider implementing a *presenter* object that is responsible for owning and displaying the form instance, especially for non-modal forms; the *Model-View-Presenter* UI pattern is well documented, and like everything OOP, its concepts aren't specific to any language or platform.
- Hide the form, don't let it self-destruct. Avoid closing and unloading.

UI Design Guidelines

I'm not going to pretend to be a guru of UI design, but over the years I've come to find myself consistently incorporating the same elements in my modal forms, and it has worked very well for me so here we go turning that into general guidelines.

Looking Fresh with Microsoft Forms

When I design a form, I usually split it in 3 parts: there's a panel at the top, another panel at the bottom, and then there's the client area. The top panel presents an icon, a title, and a label with concise instructions for the user. The bottom panel contains minimally an *OK* button, but usually I'll have *Accept* and *Cancel* buttons there. On a non-modal form there'll be a *Close* button, and if the action of the *OK* button can be executed without closing the form, I'll add an *Apply* button. The client area contains the form fields and controls with their respective labels, ideally with a balanced layout that doesn't look too much like it was hand-crafted by an amateur: that's achieved by keeping margins consistent, aligning the labels with the fields, and avoiding the use of non-system colors.

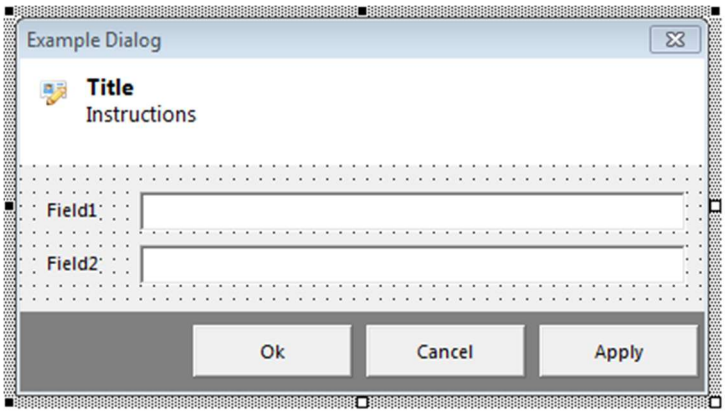

Figure 6 An example dialog user form visual design

- `TopPanel` is a `Label` control with a white background that is docked at the top and tall enough to comfortably fit short instructions.
- `BottomPanel` is also a `Label` control, with a dark gray background, docked at the bottom and no more than 32 pixels in height.
- `DialogTitle` is another `Label` control with a **bold** font, overlapping the `TopPanel` control.
- `DialogInstructions` is another `Label` control overlapping the `TopPanel` control.
- `DialogIcon` is an `Image` control for a 16×16 or 24×24 .bmp icon aligned left, at the same `Top` coordinate as the `DialogTitle` control.
- `OkButton`, `CancelButton`, `CloseButton`, `ApplyButton` would be `CommandButton` controls overlapping the `BottomPanel` control, aligned right.

Client Area

The actual client area content layout isn't exactly free-for-all, and I doubt it's possible to come up with a set of "rules" that can apply universally, but we can try, yeah?

- Identify each field with a label; align all fields to make it look like an implicit grid.
- Seek visual balance; ensure a relatively constant margin on all sides of the client area, space things out but not too much. Use `Frame` controls to group `ComboBox` options.
- Avoid making a complex form with too many responsibilities and, inevitably, too many controls. Beyond a certain complexity level, consider making separate forms instead of tabs.
- Use Segoe UI for a more modern font than the MS Sans Serif that was the standard in 1998.
- Do not **bold** all the labels.
- Have a `ToolTip` string for the label of every field the user must interact with. If a field is required or demands a particular format/pattern, indicate it.
- Consider toggling the visibility of a 16×16 icon next to (or even inside, right-aligned) input fields, to clearly indicate any data validation errors (have a tooltip string on the image control with the validation error message, e.g. "this field is required", or "value cannot be greater than 100").
- Name. All. The. Things.
- Set the `TabIndex` of every control in a logical and instinctive manner, so that the form can be easily navigated with the keyboard using the TAB key. Hitting TAB when you're in a textbox and having the focus jump to some random other control can be jarring and frustrating for users, and that's something we admittedly don't always think about!
- Use background colors in input controls only to *strongly* signal something to the user, like a validation error that must be corrected to move on. Dark red text over a light pink background makes a very strong statement for an invalid field value.
- Keep a consistent color scheme/palette and style across all your application's UI components.

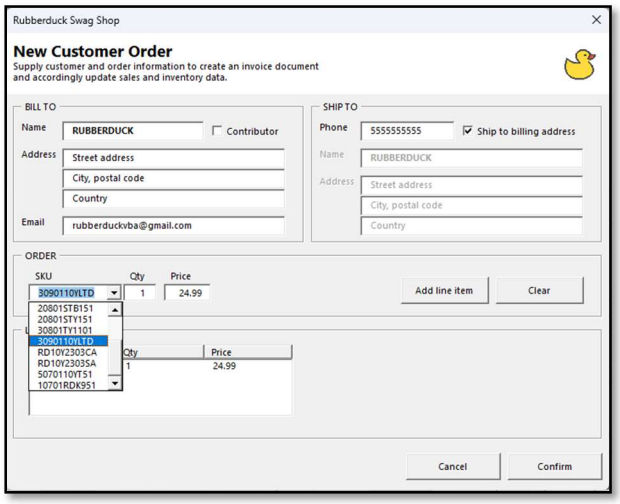

Figure 7 The order form from the swag shop's own sales and inventory tracking workbook just had to be VBA.

Extras

Extras

Architectural Patterns

When a VBA program needs a user interface, leaving the form in charge of what's going on makes the code a *Smart UI* [anti-]pattern that works nicely for a quick prototype, but does not scale well and can easily become the stuff of nightmares.

This is where OOP really starts shining in VBA, because with OOP you can break free of the mold and build your application with the very same paradigms used in "real" applications usually written in more OOP-centric languages such as VB.NET, Java, or C#.

Design patterns are somewhat of a beginner trap in object-oriented programming because it can be tempting to implement them for their own sake: these patterns are *communication tools* that signal other developers that you know what you're doing and why. They are recognizable, distinctive ways to organize the responsibilities of classes and abstractions that, in the case of *architectural patterns*, enable fully decoupling the user interface from the rest of the code, making it possible to automate thoroughly testing the application logic without having to deal with a UI to exercise every possible state.

Using these patterns obviously demands a firm grasp of concepts around classes and interfaces; as such they are advanced topics that may feel overwhelming or outright overkill; you do not *need* OOP or design patterns to achieve a working solution!

If you feel like exploring further possibilities, pushing the boundaries of what can be achieved with VBA, or writing code that can more easily be ported to another language in the future, then this is the way.

You want to think about your program in terms of interfaces & APIs, *layers* that each have their own distinct concerns, inputs, processes, and outputs. To me the interesting part is the interface/API: it's where you get to really make the code express what you want it to accomplish; the naming of each method and property, the assumptions you bake in or guard against, the errorss you raise when things go south – *that* is your API. What things look like inside the methods would typically be trivial code that's simple to wrap one's head around; 5-10, 20 lines, tops. If it's more than that, a level of abstraction or a dependency is probably missing; could the *Single Responsibility Principle* be taking a beating? Extracting things into smaller scopes, and taking the time to name them properly, often makes the abstractions simply *emerge* out of necessity – when we try to reduce repetition (*don't repeat yourself / DRY),* for example.

That's how design patterns like *Repository* and *Unit of Work* (and many others) came to be: problems that keep needing a solution tend to bring about similar abstractions every time.

In the case of *architectural patterns*, it's important to acknowledge that they are intended for *software development*, and that in the real world, most VBA projects aren't (and don't need to be) that.

But that doesn't mean they *cannot* be.

Model-View-Presenter (MVP)

In both VBA and VB6, the UI framework we work with is *Microsoft Forms* (MSForms), the direct ancestor of .NET's *Windows Forms* framework. Although the .NET version is definitively more modern and capable, both have the exact same underlying mechanisms and work very similarly. The most recommended UI design pattern for *Windows Forms* is *Model-View-Presenter* (MVP), which separates concerns in three distinct abstractions:

- The *View* is an abstraction of the form itself.
- The *Model* represents the data that the form needs to manipulate.

This works very well with *Windows Form* and has no reason to not work just as well with the older *Microsoft Forms* framework: everything you can read online about this pattern can be translated into VBA. As with other similar patterns the key is to consistently move functionality away from the UI layer: the *presenter* is responsible for communicating with the *model* layer and pass data to and from the *View*. It's very hard to make sense of it all without a very concrete example, so let's set up a hypothetical scenario where we have actual requirements: say we wanted to prompt the user for *weight* and *height*, and we wanted to compute a BMI and output it somewhere. To complicate things a bit, let's say the user can enter the measurements in either imperial or metric values (not *too* complicated; measurement system is consistent across the values, so metric height means weight is also metric).

Model

The *model* in this case consists of the user's inputs, so we already know what we need:

- HeightValue As Double
- WeightValue As Double
- UnitType As UnitType

Our model consists of a class module (let's call it `BodyMassCalcModel`) that could look like this:

```
'@Description "Represents the data needed to compute a BMI value"
Option Explicit

Public Enum MeasurementUnitType
    Metric
    Imperial
End Enum

Public HeightValue As Double
Public WeightValue As Double
Public UnitType As MeasurementUnitType
```

From there, we'll want to be able to convert back and forth to and from metric/imperial (a user might be undecisive?), so we add methods for that:

```
'@Description "Represents the data needed to compute a BMI value."
```

```
Option Explicit

Private Const CentimetersPerInch = 2.54
Private Const PoundsPerKilogram = 2.20462262

Public Enum MeasurementUnitType
    Metric
    Imperial
End Enum

Public HeightValue As Double
Public WeightValue As Double
Public UnitType As MeasurementUnitType

Public Sub ToMetric()
    If UnitType = MeasurementUnitType.Metric Then Exit Sub
    Debug.Assert UnitType = MeasurementUnitType.Imperial 'assert assumptions
    HeightValue = HeightValue * CentimetersPerInch
    WeightValue = WeightValue / PoundsPerKilogram
End Sub

Public Sub ToImperial()
    If UnitType = MeasurementUnitType.Imperial Then Exit Sub
    Debug.Assert UnitType = MeasurementUnitType.Metric 'assert assumptions
    HeightValue = HeightValue / CentimetersPerInch
    WeightValue = WeightValue * PoundsPerKilogram
End Sub
```

View

The *view* is going to be a `UserForm` class module, which comes with a *predeclared instance* so you could make it stateful and do `UserForm1.Show` if you wanted to, but with *Model-View-Presenter* showing the form alone will not do anything, because the role of a *view* isn't to *do* anything.

The *view* is an I/O device that sends and receives information to and from the user through various means. It holds a reference to and manipulates the state of the *model*, while holding a *visual state* that is consistent: the code-behind in a `UserForm` module is therefore responsible for mostly just synchronizing state between the model and what's being presented.

The form might consist of two labelled text boxes and radio buttons to select the units of measure, and then it's going to be shown modally so there'll be OK and Cancel buttons… and that means we can add a `IsCancelled As Boolean` public field to our model class to track the cancellation state of the view.

The code-behind should read like boring code that can't go wrong: we handle control events and synchronize the model properties, end of story.

```
Option Explicit
Public Model As BodyMassCalcModel

Private Sub WeightValueBox_Change()
    On Error Resume Next
```

```
    Model.WeightValue = CDbl(WeightValueBox.Value)
    On Error GoTo -1
End Sub

Private Sub HeightValueBox_Change()
    On Error Resume Next
    Model.HeightValue = CDbl(HeightValueBox.Value)
    On Error GoTo -1
End Sub

Private Sub OptionMetric_Change()
    If OptionMetric.Value Then
        Model.UnitType = MeasurementUnitType.Metric
    End If
End Sub

Private Sub OptionImperial_Change()
    If OptionImperial.Value Then
        Model.UnitType = MeasurementUnitType.Imperial
    End If
End Sub
```

The reason for temporarily disabling error handling is that we don't know and cannot assume that the inputs will be assignable to a `Double` without issues. We could implement various input validation mechanisms, but for the sake of this example resuming with the assignment if the conversion fails, is good enough: we'll just implicitly assign a zero in case of an error and move on as if nothing happened. We could consider the model invalid given a height or a weight of zero and disable the OK button then, but let's not go there for now.

What's important is that we `Hide` the current form instance (`Me`) and synchronize the model on cancellation:

```
Private Sub OkButton_Click()
    Me.Hide
End Sub

Private Sub CancelButton_Click()
    OnFormCancelled
End Sub

Private Sub OnFormCancelled()
    Model.IsCancelled = True
    Me.Hide
End Sub
```

The `QueryClose` implementation can then invoke `OnFormCancelled`, too, to synchronize cancellation state when the user closes the form by clicking the little "X" icon in the corner:

```
Private Sub UserForm_QueryClose(ByRef Cancel As Integer, ByRef CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        OnFormCancelled
    End If
```

```
End Sub
```

That's pretty much the only code that goes into the view; we're ready for the show.

Presenter

If we add a new module, name it `BodyMassCalcPresenter` (to match the name of the View and Model), we could write a `Show` macro like this:

```
Option Explicit

Public Sub Show()
    Dim Model As BodyMassCalcModel
    Set Model = New BodyMassCalcModel

    Dim View As BodyMassCalcView
    Set View = New BodyMassCalcView
    Set View.Model = Model

    View.Show
    If Model.IsCancelled Then Exit Sub

    CalculateBMI Model
End Sub

Private Sub CalculateBMI(ByVal Model As BodyMassCalcModel)
    Model.ToMetric
    Debug.Print Model.WeightValue / (Model.HeightValue ^ 2)
End Sub
```

This illustrates the pattern while also showing the problem with a mutable model: the calculation must ensure that it's working with metric values, so it runs `Model.ToMetric` to do that, but now there's a side effect: the model's state has changed, and while in this particular case it might not be a problem (it goes out of scope at the next instruction), but we'll want to design our objects in a more *encapsulated* way that doesn't expose the data directly through public fields and converts by creating and returning a brand new instance instead of mutating the internal state of the current instance.

Model-View-ViewModel (MVVM)

In .NET, MVVM works very well with *Windows Presentation Foundation* (WPF), a UI Framework that uses XAML markup to describe the *View*, and XAML *bindings* can bind the text of a text box to a `String` property of the *ViewModel* (VM), for example; buttons bind to *commands* exposed by the VM, and the framework automatically evaluates whether the button should be enabled or not depending on whether a `CanExecute` function implemented by the command returns `True` or `False`.

Property bindings can be one-way (VM-to-View), one-way to source (View-to-VM), two-way, or one-time; except for the latter, every binding mode involves a way to respond to changes in VM properties, *View* (form) control properties, or both. In .NET this is solved by implementing an `INotifyPropertyChanged` interface that mandates the presence of a `PropertyChanged` event; the binding must therefore register this event, and it's up to the programmer to write a VM that properly implements the interface and correctly propagates value changes. None of this is impossible to achieve in VBA, if we track and invoke our handlers ourselves, because abstract interfaces cannot expose an `Event` in VBA.

XAML markup gets compiled into .NET instructions; each XAML element (more or less) corresponds to a class in WPF, so in theory nothing the markup does is impossible to achieve in pure code, but the inverse isn't necessarily true. The difference is that WPF does its own rendering, and bypasses the age-old `user32` library; its capabilities go well beyond anything *Windows Forms* can do, but the learning curve is rather steep.

A "lite" version of MVVM can be achieved in VBA with just a handful of support classes to define bindings and property change notifications, and by implementing a *command manager* to manage *command bindings* you could invoke the bound command's `CanExecute` function and accordingly enable or disable the bound command button UI control; the *command manager* should be invoked when the *binding manager* (which tracks the normal data bindings to VM properties) determines that the current VM state justifies re-evaluating the `CanExecute` function for each registered command binding.

Instead of XAML markup, we must use an all-code approach where we initialize the *View* (form) by creating all the bindings explicitly… and then the only code-behind you need is code that's strictly concerned with the *presentation;* the application logic is encapsulated in *commands* that are bound to buttons, and the model data is bound to other form controls. The *Model* is the data and the means to obtain (or store) it; the VM exposes the Model data that the View needs to display – that could be the items for a combo box, for example. Ideally the VM provides a property for everything the *View* might need a binding for, so combo box values, but also the current selection and the corresponding description text; the text for the form's title, the instructions label, button captions… and now you're starting to see how much easier this approach can make something like *localization* easier to achieve: if UI captions are just data, then they're whatever we tell them to be at run-time.

Of course, in VBA we don't have a whole framework to support this, but we can still achieve MVVM by introducing a small number of key classes and interfaces to support it.

Propagating Property Changes

We've seen with MVP how we can propagate UI state to a Model instance: we handle control events, and then set the appropriate property value to synchronize the model with the UI. But what if we could also propagate state changes in the other direction? Manipulating model properties would propagate to the UI! We would need the model to expose an `Event` that would need to be raised whenever a model property changes, and the form would need to be handling this event by determining what control to affect depending on what property was modified.

The first thing to do is to take out model class and *encapsulate* its public fields into proper properties, with `Get` and `Let` accessors. If you're using Rubberduck this is just a few clicks: right-click any of the public fields and select *refactor/encapsulate field* from the Rubberduck menu, select all the fields, check the box to generate a `Private Type`, and you're done.

The `Property Let` procedures will run when we are assigning a new value to a property: that's where we'll want to raise an event to signal the property change.

So we need an `Event`. The problem is that events are an *implementation detail*, and they're only ever exposed on the default interface of a given class. This won't do, because we need a solution that doesn't need to be re-engineered from scratch every time: we need a way to encapsulate *event propagation* into its own abstraction, and then we need to manually register and invoke *callbacks* instead of raising and handling events. In .NET MVVM, the ViewModel implements `INotifyPropertyChanged`, which makes an object expose a `PropertyChanged` event. So let's have our own such interface for our ViewModel to implement, exposing method that can register a "handler", and a `NotifyPropertyChanged` method that will iterate the registered handlers and let each one know about the modified property. So first we need to define an interface that needs to be implemented by objects that can be registered as such handlers; let's call it `IHandlePropertyChanged`, and give it a `HandlePropertyChanged` method:

```
'@Interface
Option Explicit

Public Sub HandlePropertyChanged(ByVal Source As Object, ByVal Name As String)
End Sub
```

Now we have defined the handlers, we can define `INotifyPropertyChanged` as follows:

```
'@Interface
Option Explicit

Public Sub AddHandler(ByVal Handler As IHandlePropertyChanged)
End Sub

Public Sub OnPropertyChanged(ByVal Source As Object, ByVal Name As String)
End Sub
```

We know we'll want our ViewModel to implement the notifications, but then something is missing: we need an abstraction that defines a *property binding*, and that is the object that will be on the receiving end of these notifications: the binding will also be listening for control events, but because each type of control has its own set of events, we should have classes like `TextBoxValueBinding` that implement `IHandlePropertyChanged` to propagate ViewModel changes to the UI:

```
Option Explicit
Implements IHandlePropertyChanged
Private WithEvents UI As MSForms.TextBox

Private Type TBinding
    Source As Object
    SourceProperty As String
End Type
Private This As TBinding

Public Sub Initialize(ByVal Target As MSForms.TextBox, ByVal Source As Object, ByVal
SourceProperty As String)
    Set UI = Target
    Set This.Source = Source
    This.SourceProperty = SourceProperty
    If TypeOf Source Is INotifyPropertyChanged Then RegisterPropertyChanges Source
End Sub

Private Sub RegisterPropertyChanges(ByVal Source As INotifyPropertyChanged)
    Source.AddHandler Me
End Sub

Private Sub IHandlePropertyChanged_HandlePropertyChanged(ByVal Source As Object, ByVal
Name As String)
    If Source Is This.Source And Name = This.SourceProperty Then
        UI.Text = VBA.Interaction.CallByName(This.Source, This.SourceProperty, VbGet)
    End If
End Sub

Private Sub UI_Change()
    VBA.Interaction.CallByName This.Source, This.SourceProperty, VbLet, UI.Value
End Sub
```

And then a `CheckBoxValueBinding` would be implemented very similarly:

```
Option Explicit
Implements IHandlePropertyChanged
Private WithEvents UI As MSForms.CheckBox

Private Type TBinding
    Source As Object
    SourceProperty As String
End Type
Private This As TBinding

Public Sub Initialize(ByVal Target As MSForms.CheckBox, ByVal Source As Object, ByVal
SourceProperty As String)
```

```
    Set UI = Target
    Set This.Source = Source
    This.SourceProperty = SourceProperty
    If TypeOf Source Is INotifyPropertyChanged Then RegisterPropertyChanges Source
End Sub

Private Sub RegisterPropertyChanges(ByVal Source As INotifyPropertyChanged)
    Source.AddHandler Me
End Sub

Private Sub IHandlePropertyChanged_HandlePropertyChanged(ByVal Source As Object, ByVal
Name As String)
    If Source Is This.Source And Name = This.SourceProperty Then
        UI.Text = VBA.Interaction.CallByName(This.Source, This.SourceProperty, VbGet)
    End If
End Sub

Private Sub UI_Change()
    VBA.Interaction.CallByName This.Source, This.SourceProperty, VbLet, UI.Value
End Sub
```

Since we don't have any other control types in our UI, there's no need to implement binding classes for every possible type of MSForms control: the bindings all look alike and perform similar duties, implementing a new one when we need it shouldn't the too complicated, and then a `OneWayPropertyBinding` would also be useful:

```
Option Explicit
Implements IHandlePropertyChanged

Private Type TBinding
    InvertBoolean As Boolean
    Source As Object
    SourceProperty As String
    Target As Object
    TargetProperty As String
End Type

Private This As TBinding

Public Sub Initialize(ByVal Target As MSForms.Control, ByVal TargetProperty As String,
ByVal Source As Object, ByVal SourceProperty As String, Optional ByVal InvertBoolean
As Boolean = False)
    Set This.Source = Source
    This.SourceProperty = SourceProperty
    Set This.Target = Target
    This.TargetProperty = TargetProperty
    This.InvertBoolean = InvertBoolean
    If TypeOf Source Is INotifyPropertyChanged Then RegisterPropertyChanges Source
    IHandlePropertyChanged_OnPropertyChanged Source, SourceProperty
End Sub

Private Sub RegisterPropertyChanges(ByVal Source As INotifyPropertyChanged)
    Source.RegisterHandler Me
```

```
End Sub

Private Sub IHandlePropertyChanged_HandlePropertyChanged(ByVal Source As Object, ByVal
Name As String)
    If Source Is This.Source And Name = This.SourceProperty Then
        Dim Value As Variant
        Value = VBA.Interaction.CallByName(This.Source, This.SourceProperty, VbGet)
        If VarType(Value) = vbBoolean And This.InvertBoolean Then
            VBA.Interaction.CallByName This.Target, This.TargetProperty, VbLet, Not
CBool(Value)
        Else
            VBA.Interaction.CallByName This.Target, This.TargetProperty, VbLet, Value
        End If
    End If
End Sub
```

We still need a bit more infrastructure before we can configure these bindings; the View is going to want a nice expressive API to do this, so let's add a `PropertyBindingViewHelper` module:

```
Option Explicit

'@Description "Binds a MSForms.Control property to a source property"
Public Function BindProperty(ByVal Control As MSForms.Control, ByVal ControlProperty
As String, ByVal SourceProperty As String, ByVal Source As Object, Optional ByVal
InvertBoolean As Boolean = False) As OneWayPropertyBinding
    Dim Binding As OneWayPropertyBinding
    Set Binding = New OneWayPropertyBinding
    Binding.Initialize Control, ControlProperty, Source, SourceProperty, InvertBoolean
    Set BindProperty = Binding
End Function

'@Description "Binds the Text/Value of a MSForms.TextBox to a source property"
Public Function BindTextBox(ByVal Control As MSForms.TextBox, ByVal SourceProperty As
String, ByVal Source As Object) As TextBoxValueBinding
    Dim Binding As TextBoxValueBinding
    Set Binding = New TextBoxValueBinding
    Binding.Initialize Control, Source, SourceProperty
    Set BindTextBox = Binding
End Function

'@Description "Binds the Value of a MSForms.CheckBox to a Boolean source property"
Public Function BindCheckBox(ByVal Control As MSForms.CheckBox, ByVal SourceProperty
As String, ByVal Source As Object) As CheckBoxValueBinding
    Dim Binding As CheckBoxValueBinding
    Set Binding = New CheckBoxValueBinding
    Binding.Initialize Control, Source, SourceProperty
    Set BindCheckBox = Binding
End Function
```

And then implementing `INotifyPropertyChanged` comes with a bit of boilerplate code that we'll want to avoid repeating in every ViewModel we ever implement, so we add a `PropertyChangedHelper` class:

```
Option Explicit
Private Handlers As VBA.Collection

Public Sub AddHandler(ByVal Handler As IHandlePropertyChanged)
    Handlers.Add Handler
End Sub

Public Sub Notify(ByVal Source As Object, ByVal Name As String)
    Dim Handler As IHandlePropertyChanged
    For Each Handler In Handlers
        Handler.OnPropertyChanged Source, Name
    Next
End Sub

Private Sub Class_Initialize()
    Set Handlers = New VBA.Collection
End Sub
```

Back in the ViewModel, we can now implement `INotifyPropertyChanged` by modifying all `Property Let` procedures to, well, notify [only when] when the value changes:

```
'@Description "Represents the data needed to compute a BMI value."
Option Explicit
Implements INotifyPropertyChanged
Private PropertyChanges As New PropertyChangedHelper

Public Enum MeasurementUnitType
    Metric = 0
    Imperial = 1
End Enum

Private Const CentimetersPerInch = 2.54
Private Const PoundsPerKilogram = 2.20462262

Private Type TModel
    HeightValue As Double
    WeightValue As Double
    UnitType As MeasurementUnitType
End Type
Private This As TModel

Public Property Get HeightValue() As Double
    HeightValue = This.HeightValue
End Property

Public Property Let HeightValue(ByVal RHS As Double)
    If This.HeightValue <> RHS Then
        This.HeightValue = RHS
        OnPropertyChanged "HeightValue"
    End If
End Property
```

```
    Public Property Get WeightValue() As Double
        WeightValue = This.WeightValue
    End Property

    Public Property Let WeightValue(ByVal RHS As Double)
>       If This.WeightValue <> RHS Then
            This.WeightValue = RHS
>           OnPropertyChanged "WeightValue"
        End If
    End Property

    Public Property Get UnitType() As MeasurementUnitType
        UnitType = This.UnitType
    End Property

    Public Property Let UnitType(ByVal RHS As MeasurementUnitType)
>       If UnitType <> This.UnitType Then
            This.UnitType = RHS
>           OnPropertyChanged "UnitType"
        End If
    End Property

    Public Property Get UnitTypes() As Variant
        Static Result() As Variant
        If IsEmpty(Result) Then
            ReDim Result(0 To 1, 0 To 1)
            Result(MeasurementUnitType.Metric, 0) = MeasurementUnitType.Metric
            Result(MeasurementUnitType.Metric, 1) = "Metric"
            Result(MeasurementUnitType.Imperial, 0) = MeasurementUnitType.Imperial
            Result(MeasurementUnitType.Imperial, 1) = "Imperial"
        End If
        UnitTypes = Result
    End Property

    Public Sub ToMetric()
        If UnitType = MeasurementUnitType.Metric Then Exit Sub
        Debug.Assert UnitType = MeasurementUnitType.Imperial 'assert assumptions
>       HeightValue = HeightValue * CentimetersPerInch
        WeightValue = WeightValue / PoundsPerKilogram
    End Sub

>   Public Sub ToImperial()
        If UnitType = MeasurementUnitType.Imperial Then Exit Sub
        Debug.Assert UnitType = MeasurementUnitType.Metric 'assert assumptions
        HeightValue = HeightValue / CentimetersPerInch
        WeightValue = WeightValue * PoundsPerKilogram
>   End Sub

    Private Sub OnPropertyChanged(ByVal Name As String)
        INotifyPropertyChanged_OnPropertyChanged Me, Name
    End Sub

    Private Sub INotifyPropertyChanged_AddHandler(ByVal Handler As IHandlePropertyChanged)
        PropertyChanges.AddHandler Handler
    End Sub
```

```
Private Sub INotifyPropertyChanged_OnPropertyChanged(ByVal Source As Object, ByVal
Name As String)
    PropertyChanges.Notify Source, Name
End Sub
```

At this point everything is in place already: we just need the View to wire it all up, and this is where MVVM is really going to start shining: because the bindings handle both the control events and VM changes, there is no need to handle anything other than `QueryClose` in the form's code-behind (and the button clicks, but let's ignore them for now). This *really* cleans up the code-behind! But where to initialize and configure the bindings? If we do it in `UserForm_Initialize`, the model instance won't be set and we'll run into problems. If we do it in `UserForm_Activate`, we'll run into possible multiple initialization issues when the user clicks away and then comes back to our form. What's missing is an `IView` interface that would provide a method to inject the model and show the form to the user:

```
'@Interface IView
Option Explicit

Public Sub Show(ByVal Model As Object)
End Sub
```

And now we can implement it in the form's code-behind – the interface receives an `Object` model to be generic, but we can *cast* it to a more specific type by passing it to another method:

```
Private PropertyBindings As New VBA.Collection

Private Sub IView_Show(ByVal Model As Object)
    ConfigureBindings Model
    Me.Show
End Sub

Private Sub ConfigureBindings(ByVal Model As BodyMassCalcModel)
    Set This.Model = Model
    With PropertyBindings
        .Add PropertyBindingViewHelper.BindTextBox(Me.HeightValueBox, "HeightValue",
This.Model)
        .Add PropertyBindingViewHelper.BindTextBox(Me.WeightValueBox, "WeightValue",
This.Model)
        .Add PropertyBindingViewHelper.BindComboBox(Me.UnitTypeBox, "UnitType",
This.Model)
        .Add PropertyBindingViewHelper.BindProperty(Me.UnitTypeBox, "List",
"UnitTypes", This.Model)
    End With
End Sub
```

Once we've created the bindings and added them to an instance-level collection to keep these objects in scope, we're done: the form controls will initialize with whatever values are held by the model at that time, and then the model properties will update themselves when the user enters the input values.

And now we can talk about binding those command buttons.

Command Bindings

WPF defines an `ICommand` interface that exposes two methods: `CanExecute` should return `True` if the command can be executed with the current state, and `False` otherwise so that the framework knows to disable the bound command button – and then of course an `Execute` method that runs the command. We can define an identical interface in VBA:

```
'@Interface ICommand
Option Explicit

Public Function CanExecute(ByVal Parameter As Object) As Boolean
End Function

Public Sub Execute(ByVal Parameter As Object)
End Sub
```

We'll want two implementations for now: a `CancelCommand` that can cancel our modal form, and a `CalculateBMICommand` where we will be consuming the model and outputting the result of the calculation. Cancellation should always be an option, so the `CanExecute` logic for the `CancelCommand` is simply to return `True`, but calculating should be disabled until we have valid inputs.

We could pass the `UserForm` instance as the `CancelCommand.Execute` parameter, but then we wouldn't have access to the implementation-specific cancellation logic, just `UserForm.Hide`. And if we passed an instance of `BodyMassCalcView`, well then the command wouldn't be decoupling anything! One solution could be to expose the cancellation logic via `IView`, but then that's a very high-abstraction interface and not all implementations are going to need this, so a better solution that's aligned with the *Interface Segregation Principle* would be to introduce and implement a new `ICancellable` interface exposing a single `Cancel` method:

```
'@Interface ICancellable
Option Explicit

Public Sub Cancel()
End Sub
```

To implement it, we simply make `ICancellable_Cancel` invoke the private `OnCancel` method, and we're done: now the cancel command can receive an `ICancellable` object as a parameter, and invoke its `Cancel` method, which will correctly affect the cancellation state in the model. And since the command isn't tied to any specific class, it can be reused with anything that ever implements `ICancellable`!

```
Implements ICommand

Private Function ICommand_CanExecute(ByVal Parameter As Object) As Boolean
    ICommand_CanExecute = True
End Function
```

```
Private Sub ICommand_Execute(ByVal Parameter As Object)
    Dim Cancellable As ICancellable
    If TypeOf Parameter Is ICancellable Then Set Cancellable = Parameter
    If Cancellable Is Nothing Then Exit Sub
    Cancellable.Cancel
End Sub
```

And then the View can implement `ICancellable` by invoking its cancellation logic:

```
Implements IView
Implements ICancellable

Private Sub OnFormCancelled()
    Model.IsCancelled = True
    Me.Hide
End Sub

Private Sub ICancellable_Cancel()
    OnFormCancelled
End Sub
```

The calculation command is more specific to this project/example, and that's fine: not all commands need to be general-purpose high-abstraction stuff! Since this command needs the model, we'll be receiving the `BodyMassCalcModel` instance for its parameter:

```
Implements ICancellable

Private Sub ICancellable_Cancel()
    OnFormCancelled
End Sub
```

Now that we have commands, we need to wire them up and bind them to command buttons in the UI. We can use (one-way) property bindings to control their captions and tooltips, but we need something that will listen to the `Click` event and respond by executing the appropriate command. Enter the `CommandBinding`:

```
Option Explicit
Private WithEvents UI As MSForms.CommandButton

Private Type TCmdBinding
    Command As ICommand
    Parameter As Object
End Type
Private This As TCmdBinding

Public Sub Initialize(ByVal Command As ICommand, ByVal Parameter As Object)
    Set This.Command = Command
    Set This.Parameter = Parameter
End Sub

Public Sub EvaluateCanExecute()
    This.Button.Enabled = This.Command.CanExecute(This.Parameter)
```

```
End Sub

Private Sub UI_Click()
    This.Command.Execute This.Parameter
End Sub
```

There's one problem left to address: manipulating the model will fire `PropertyChanged` handlers, but none of them will trigger a call to the commands' `CanExecute` functions! Having a collection of property bindings isn't sufficient, we need a `BindingManager` (for lack of a better name) object that knows about the command and property bindings and can *also* listen for property changes and accordingly re-evaluate whether commands can be executed given the new state of the model, by invoking `EvaluateCanExecute` on each command.

```
Option Explicit
Implements IHandlePropertyChanged
Private PropertyBindings As VBA.Collection
Private CommandBindings As VBA.Collection

Private Class_Initialize()
    Set PropertyBindings = New VBA.Collection
    Set CommandBindings = New VBA.Collection
End Sub

Public Sub AddPropertyBinding(ByVal Binding As Object)
    PropertyBindings.Add Binding
    If TypeOf Binding Is INotifyPropertyChanged Then
        Dim NotifyingBinding As INotifyPropertyChanged
        Set NotifyingBinding = Binding
        NotifyingBinding.AddHandler Me
    End If
End Sub

Public Sub AddCommandBinding(ByVal Binding As CommandBinding)
    CommandBindings.Add Binding
End Sub

Private Sub IHandlePropertyChanged_HandlePropertyChanged(ByVal Source As Object, ByVal
Name As String)
    Dim Binding As CommandBinding
    For Each Binding In CommandBindings
        Binding.EvaluateCanExecute Source
    Next
End Sub
```

And now the View can have a `Private Bindings As BindingManager` private field, and by registering the `BindingManager` instance as a handler for a binding's notifications, we get to respond to them and evaluate whether commands can be executed.

All that's left to do is to implement `INotifyPropertyChanged` in the binding classes to relay property changes to the manager!

As you can see MVVM demands quite a bit of infrastructure to work, and the more features we want it to support, the more infrastructure code we're going to need: MVVM really wants to be a *library* that defines all this infrastructure, so we only need to use it.

It's a fun exercise though and making MVVM work in VBA is extremely rewarding. Some implementation details in the above BMI calculator project have been left for the reader to complete.

Conclusions

Because of the necessary boilerplate and despite how much fun MVVM to work with, the go-to UI design pattern in VBA should be MVP / Model-View-Presenter. It can be implemented without any boilerplate, with a minimal model that doesn't even need to encapsulate public fields… although, it arguably should.

For academic/learning purposes however, MVVM makes a great pretext for learning many techniques.

The purpose of these patterns is to decouple the model and the logic from the view, so that the model and the logic can be tested *without* the view.

Whether or not there are actual unit tests covering that code, what matters is that it is at least *testable*.

Unit Testing

A unit test is a small, simple procedure that is responsible for 3 things:

1. **Arrange** dependencies and set expectations.
2. **Act**, by invoking the method or function under test.
3. **Assert** that the expected result matches the actual one.

That's why the template for a Rubberduck test method looks like this:

```
'@TestMethod("Uncategorized")
Private Sub TestMethod1() 'TODO Rename test
    On Error GoTo TestFail

    'Arrange:

    'Act:

    'Assert:
    Assert.Succeed

    TestExit:
    '@Ignore UnhandledOnErrorResumeNext
    On Error Resume Next

    Exit Sub
    TestFail:
    Assert.Fail "Test raised an error: #" & Err.Number & " - " & Err.Description
    Resume TestExit
End Sub
```

Snippet 1 The standard test method template ensures any errors raised during its execution fail the test with a relatively friendly error message. The Succeed call is only present to ensure that a test always executes at least one Assert member call, otherwise the test would be inconclusive. It's the @TestMethod annotation (and the @TestModule annotation at the top of a standard module) that makes a procedure discoverable as a Rubberduck unit test.

When a unit test runs, Rubberduck tracks `Assert.Xxxx` method calls (in bold above) and their outcome; if a single `Assert` call fails (the template suggests and encourages a single assert in the normal execution path, but there can be multiple ones), the test fails. Such automated tests are very useful to document the requirements of a particular *model* class, or the behavior of a given utility function with multiple optional parameters. With enough *coverage*, tests can actively prevent *regression bugs* from being inadvertently introduced as the code is maintained and modified: if a tweak breaks a test, you know exactly what functionality you broke, and if all tests are green you know the code is still going to behave as intended.

Have a test module per unit/class you're testing, and consider naming the test methods following a `MethodUnderTest_GivenAbcThenXyz`, where `MethodUnderTest` is the name of the method you're testing, `Abc` is a particular condition, and `Xyz` is the expected outcome. For tests that expect an error, consider following a `MethodUnderTest_GivenAbc_Throws` naming pattern. Rubberduck will not warn about underscores in test method names, and these underscores are safe because Rubberduck test modules

are standard modules, and unit test naming recommendations usually heavily favor being descriptive over being concise.

What to test?

You want to test each object's *public interface*, and treat an object's `Private` members as *implementation details*. You do NOT want to test *implementation details*. For example if a class' *default interface* only exposes a handful of `Property Get` members and a `Create` factory method that performs property-injection and a handful of properties, then in order to test that factory method there should be tests that validate that each of the parameters of the `Create` method correspond to an injected property. If one of the parameters isn't allowed to be `Nothing`, then there should be a *guard clause* in the `Create` method for it, and a unit test that ensures a specific error is being raised when the `Create` method is invoked with `Nothing` for that parameter.

Below is one such simple example, where we have 2 properties and a method; note how tests for the private `InjectDependencies` function would be redundant if the public `Create` function is already covered; the `InjectDependencies` function is an *implementation detail* of the `Create` function:

```
'@PredeclaredId

Option Explicit

Implements IClass1

Private Type TState
    SomeValue As String
    SomeDependency As Object
End Type
Private This As TState

Public Function Create(ByVal SomeValue As String, ByVal SomeDependency As Object) As
IClass1
    If SomeValue = vbNullString Then Err.Raise 5
    If SomeDependency Is Nothing Then Err.Raise 5
    Dim Result As Class1
    Set Result = New Class1
>   InjectProperties Result, SomeValue, SomeDependency
    Set Create = Result
End Function

Private Sub InjectProperties(ByVal Instance As Class1, ByVal SomeValue As String,
ByVal SomeDependency As Object)
    Instance.SomeValue = SomeValue
    Set Instance.SomeDependency = SomeDependency
End Sub

Public Property Get SomeValue() As String
    SomeValue = This.SomeValue
End Property

Public Property Let SomeValue(ByVal RHS As String)
```

```
    This.SomeValue = RHS
End Property

Public Property Get SomeDependency() As Object
    SomeDependency = This.SomeDependency
End Property

Public Property Set SomeDependency(ByVal RHS As Object)
    Set This.SomeDependency = RHS
End Property

Private Property Get IClass1_SomeValue() As String
    IClass1_SomeValue = This.SomeValue
End Property

Private Property Get IClass1_SomeDependency() As Object
    IClass1_SomeDependency = This.SomeDependency
End Property
```

What is testable?

Without the `Property Get` members of `Class1` and/or `IClass1`, we wouldn't be able to test that the `Create` method is property-injecting `SomeValue` and `SomeDependency`, because the object's internal state is ***encapsulated*** (as it should be). Therefore, there's an implicit assumption that a `Property Get` member for a property-injected dependency is returning the encapsulated value or reference, and nothing more: by writing tests that rely on that assumption, we are documenting it. Note that this level of coverage might be very much *overkill* for your purposes; there's a level of *reasonable assumptions* that's perfectly fine to have, not every property of every class needs to be fully covered!

Now `SomeDependency` might be an instance of another class, and that class might have its own encapsulated state, dependencies, and testable logic. A more meaty `Class1` module might have a method that invokes `SomeDependency.DoSomething`, and the tests for that method would have to be able to assert that `SomeDependency.DoSomething` has been invoked once.

If `Class1` wasn't property-injecting `SomeDependency` (for example if `SomeDependency` was being `New`'d up instead), we would not be able to write such a test, because the outcome of the test might be dependent on a method being called against that dependency.

A simple example would be `Class1` newing up a `FileSystemObject` to iterate the files of a given folder. In such a case, `FileSystemObject` is a dependency, and if `Class1.DoSomething` is newing it up directly then every time `Class1.DoSomething` is called, it's going to try and iterate the files of a given folder, because that's what a `FileSystemObject` does, it hits the file system. And that's slow. I/O (file, network, …and user) is dependent on so many things that can go wrong for so many reasons, having it interfere with tests is something you will generally want to avoid.

The way to avoid having user, network, and file inputs and outputs interfere with the tests of any method, is to ***completely let go*** of the "need" for a method to ***control*** any of its dependencies. The method doesn't

need to create a new instance of a `FileSystemObject`; what it really needs is actually a much simpler *any object that's capable of returning a list of files or file names in a given folder*.

So instead of creating a `FileSystemObject` dependency directly like this:

```
Public Sub DoSomething(ByVal Path As String)
    With CreateObjet("Scripting.FileSystemObject")
        ' gets the Path folder...
        ' iterates all files...
        ' ...
    End With
End Sub
```

We would instead have a dependency on an abstraction (interface) that we could inject at the instance level if it makes sense, or at the method level by taking it in as a parameter:

```
Public Sub DoSomething(ByVal Path As String, ByVal FileProvider As IFileProvider)
    Dim Files As Variant
    Files = FileProvider.GetFiles(Path)
    ' iterates all files...
    ' ...
End Sub
```

Where `IFileProvider` would be an interface/class module that might look like this:

```
'@Interface
Option Explicit

Public Function GetFiles(ByVal Path As String) As Variant
End Function
```

That interface might very well be implemented in a class module named `FileProvider` that uses a `FileSystemObject` to return the promised array:

```
Option Explicit
Implements IFileProvider

Private Function IFileProvider_GetFiles(ByVal Path As String) As Variant
    With CreateObject("Scripting.FileSystemObject")
        Dim Folder As Object
        Set Folder = .GetFolder(Path)
        ReDim Result(1 To Folder.Files.Count)

        Dim File As Object, i As Long
        For Each File In Folder.Files
            i = i + 1
            Result(i) = File.Name
        Next
    End With
    IFileProvider_GetFiles = Result
End Function
```

Thanks to *polymorphism* it could also be implemented in another class module named `TestFileProvider`, that uses a `ParamArray` parameter so that unit tests can *take control* of the `IFileProvider` dependency and *inject* (here by *method injection*) a `TestFileProvider` instance. The `DoSomething` method doesn't need to know where the file names came from, only that it can expect an array of existing, valid file names from `IFileProvider.GetFiles(String)`. If the `DoSomething` method indeed doesn't care where the files came from, then it's adhering to pretty much *all* OOP design principles, and now a test can be written that fails if `DoSomething` is *doing something* wrong – as opposed to a test that might fail if some network drive happens to be dismounted or works locally when working from home but only with a VPN.

The hard part is obviously *identifying the dependencies* in the first place. If you're refactoring a procedural VBA macro, you must determine what the inputs and outputs are, what objects hold the state that's being altered, and devise a way to abstract them away and inject these dependencies from the calling code – whether that caller is the original entry point macro procedure, or a new unit test.

Mocking vs Stubbing

In the above example, the `TestFileProvider` implementation of the `IFileProvider` dependency is essentially a *test stub*: you actually write a separate implementation for the sole purpose of being able to run the code with fake dependencies that *don't* incur any file, network, or user I/O. Reusing these stubs in "test" macros that wire up the UI by injecting the test stubs instead of the actual implementations, should result in the application running normally... without hitting any file system or network.

With *mocks*, you don't need to write a "test" implementation. Instead, you configure an object provided by a *mocking framework* to behave as the method/test needs, and the framework implements the mocked interface with an object that can be injected, that verifiably behaves as configured.

Sounds like magic? A lot of it actually is, from a VBA/VB6 standpoint. Many tests in Rubberduck leverage a very popular mocking framework called **Moq**. What we're going to be releasing as an *experimental feature* is not only a COM-visible wrapper around Moq. The fun part is that the Moq methods we need to use are generic methods that take *lambda expressions* as parameters, so our wrapper needs to expose an API VBA code can use, and then "translate" it into member calls into the Moq API, but because they're generic methods and the mocked interface is a COM object, we essentially build a .NET type on the fly to match the mocked VBA/COM interface, so that's what Moq actually mocks: a .NET interface type Rubberduck makes up at run-time from any COM object. Moq uses the Castle Windsor library under the hood to spawn instances of *proxy types* – made-up actual objects that actually implement one or more interfaces. Castle Windsor is excellent at what it does; we use CW to automate *dependency injection* in Rubberduck (a technique dubbed *Inversion of Control*, where a single *container* object is responsible for creating all instances of all objects in the application in the *composition root*; that's what's going on while Rubberduck's splash screen is being displayed).

There is a problem though: CW seems to be caching types with the reasonable but still implicit assumption that the type isn't going to change at run-time. In our case however, this means mocking a VBA interface once and then modifying that interface (e.g. adding, removing, or reordering members, or changing a member signature in any way) and re-running the test would still be mocking the old interface, as long as the host process lives. This isn't a problem for mocking a `Range` or a `Worksheet` dependency, but VBA user code is being punished here.

Verifiabale Invocations

Going back to the `IFileProvider` example, the `GetFiles` method could be configured to return a hard-coded array of bogus test strings, and a test could be made to turn green when `IFileProvider.GetFiles` is invoked with the same specific `Path` parameter value that was given to `Class1.DoSomething`. If you were *stubbing* `IFileProvider`, you would perhaps increment a counter every time `IFileProvider_GetFiles` is invoked, and expose that counter with a property that the test could `Assert` is equal to an expected value. With Moq, you can make a test fail by invoking a `Verify` method on the mock itself, that ***verifies*** whether the specified method was invoked as configured.

A best practice with mocking would be to only setup the minimal amount of members to make the test work, because of the performance overhead: if a mocked interface has 5 methods and 3 properties but the method under test only needs 2 of these methods and 1 of these properties, then it should only setup these. Verification makes mocking a very valuable tool to test behavior that relies on side-effects and state changes.

The best part is that because VBA is COM, then ***everything is an interface***, so if you don't have an `IFileProvider` interface but you're still passing a `FileProvider` object as a dependency, then you can mock the `FileProvider` directly and don't need to introduce any extra "just-for-testing" `IFileProvider` interface if you don't already have one.

Rubberduck Fakes API

While actual mocking has been encountering issues, Rubberduck unit tests have yet another tool available, that can make a difference and help reduce the number of wrappers and test stubs that need to be implemented in order to be able to test things without having `MsgBox` interfere.

Indeed, during the execution of a unit test, Rubberduck can be instructed to hook into the VBA run-time itself, intercept some specific internal function calls, and hijack their return values to make them behave any way your test needs it to. So you can write a test for a method that prompts the user whether or not to proceed with a certain action, instruct the `MsgBox` function to return `VbMsgBoxResult.vbYes` when invoked, and then assert that the action was executed – or, instruct it to return `VbMsgBoxResult.vbNo`, and assert the opposite. The *Fakes API* works similarly to Moq, in the sense that you can configure any of the hijacked functions to return the needed values given such or such parameterization and/or for a specific invocation, and then you can `Verify` each invocation against the *fake* wrapper.

Severity Levels

In Rubberduck each inspection has a configurable *severity level* that defaults to `Warning` for most inspections (it's the *default-unless-specified-otherwise* for all Rubberduck inspections):

- `Error` level indicates a potential problem you likely want to pay immediate attention to, because it could be (or cause) a bug. If inspection results rendered in the code pane (they will in RD3), these would be red squiggly underlines screaming for attention.
- `Warning` level indicates a potential issue you should be aware of.
- `Suggestion` level is usually used for various opportunities, not necessarily problems.
- `Hint` level is also for various non-problematic opportunities. If inspection results rendered in the code pane, these would be a subtle dotted underline with a hover text.
- `DoNotShow` disables the inspection: not only its results won't be shown; they won't even be *generated*.

By default, Rubberduck is configured to run all (that's currently over 100) inspections, with a handful of cherry-picked exceptions for inspections that would be flagging the exact opposite situation that another enabled inspection is already flagging – for example we ship *implicit ByRef modifier* enabled (as a `Hint`), but redundant ByRef modifier is disabled unless you give it a severity level other than `DoNotShow`. This avoids "fixing" one inspection result only to get a new one flagging the exact opposite, which would be understandably confusing for users that aren't familiar with static code analysis tooling.

Are inspections somehow imbued with the knowledge of whether you should treat them as errors, warnings, or mere hints and suggestions? Sometimes, yes. *Missing Option Explicit* should make a clear consensus at `Error` level. On the flipside, whether an *implicit default member call* or the use of an *empty string literal* should be a `Warning`, a `Hint`, or whether it should be reported at all probably depends more on how comfortable or experienced you are with VBA/VB6 as a language, or could be just a personal preference; what matters is that the static code analysis tooling is letting you know something about the code, that the code alone isn't necessarily saying.

Severity levels should be carefully reviewed to match your style and preferences. If you're unsure about an inspection result, be sure to read about it on the website[4].

[4] https://rubberduckvba.com

Design Patterns

This section covers a small fraction of common design patterns and their respective implementation in VBA. Rubberduck is not necessary to implement any of these, but the tooling is warmly recommended.

Errors & Guard Clauses

Not exactly a *design pattern* per se, but when you validate your parameters as the fist few executable instructions in a procedure, and throw an error to *fail early* instead of carrying on and failing perhaps much later in a way that could be much more difficult to debug. These instructions are called *guard clauses* and their purpose is specifically to *fail early* to guard against programming errors and invalid inputs by eliminating assumptions.

The hard part is to identify the assumptions that are baked into our code – that is, to anticipate the various possible ways our inputs could be wrong, as if the calling code was hostile and actively trying to get our code to run with invalid inputs. If we're getting an object reference, we should raise an error if we get `Nothing`. If we're pulling the current `Selection`, we check that it's actually a `Range` object before we treat it as such. If we receive a `Range`, we check that it represents a single `Area` and has the expected shape (rows, columns), or that it represents a single cell if that's how our code expects it. If we take a numeric value, we should validate that it's within the expected range of valid values; if zero or negative values are unexpected, raise an error as soon as you encounter it. Dates that can't be in the future, strings that can't be empty, undefined `Enum` values, invocations of an instance member off the default instance of a class when that particular instance is supposed to remain stateless, etc.

Having a Guard standard module with Sub procedures like `NotNothing`, `NotEmpty`, `NotInRange`, `NotZero`, `NotNegative`, etc., that raise a run-time error given a failed check, avoids repeating these checks everywhere in perhaps varying formulations, and reads rather nicely.

```
    Public Sub DoSomething(ByVal Path As String, ByVal FileProvider As IFileProvider)
>       Guard.NotEmpty Path
>       Guard.NotNothing FileProvider
        '...
    End Sub
```

That module could expose a `Public Enum` that declares a named constant for each custom error raised by that module, so instead of comparing `Err.Number` against a hard-coded integer literal, error handlers can compare it against named constants and read like `GuardClause.ErrNotNothing` instead of some magic number. Since you're probably going to need to raise other run-time errors from other modules, keep in mind the `Enum` values to avoid overlapping!

```
Option Explicit

Public Enum GuardClause
    ErrNotNothing = vbObjectError + 1234
    ErrNotEmpty
    ErrNotZero
    '...
End Enum

Public Sub NotNothing(ByVal Value As Object)
    If Value Is Nothing Then Err.Raise GuardClause.ErrNotNothing
End Sub

Public Sub NotEmpty(ByVal Value As String)
    If Len(Value) = 0 Then Err.Raise GuardClause.ErrNotEmpty
End Sub

Public Sub NotZero(ByVal Value As Double)
    If Value = 0 Then Err.Raise GuardClause.ErrNotZero
End Sub
```

There could also be an `Errors` module that could be used for raising custom errors as needed; doing so centralizes all the custom errors raised in your application and makes it much easier to avoid overlapping and to leverage the error-trapping mechanisms in VBA for your purposes.

When to raise a custom error?

If you let VBA code run without validating the inputs, you may end up having to debug a division by zero somewhere down in the catacombs of a module, whereas guarding against invalid inputs allows you to *fail early* and *deterministically*: you know exactly why things are failing at this point in the code, and where the bad input came from is in your call stack if you break there: custom errors are there to eliminate the assumptions a procedure can make, even If they seem obvious or useless. You raise a custom error when the current *state* of the program is unexpected and there's no recovery: the calling code can then decide to handle the error, or let it bubble up the call stack to its own caller (the infamous VBA debug prompt appears when the error reaches the top of the call stack).

Private State

If you've been following my work for any amount of time, you've come across this `Private Type TInstanceState` and `Private This As TInstanceState`, where I define a single, private instance field with a UDT data type that defines all the backing fields for the class properties.

```
  Option Explicit

  Private Type TInstanceState
      SomeValue As String
  End Type
> Private This As TInstanceState

  Public Property Get SomeValue() As String
>     SomeValue = This.SomeValue
  End Property

  Public Property Let SomeValue(ByVal Value As String)
>     This.SomeValue = Value
  End Property
```

Adopting this style/pattern cleans up the *locals* toolwindow in the debugger by shoving the entire private state under the `This` declaration (or whatever you named it). It also has the benefit of making properties crystal-clear by mapping identical identifier names without conflicting (the UDT fields must be qualified).

Rubberduck can implement this pattern for you, via the *Encapsulate Field* refactoring. In your new class module, define a properly named and typed public field (could be implicitly so, doesn't matter) for each property you want to end up with; parse the modifications, then right-click any of them and select refactor/encapsulate field from the Rubberduck menu.

Tick the checkbox to create the private UDT, select all the fields, and watch your class write itself instantly; since Rubberduck modifies the code, it automatically processes the code changes.

Factories & Providers

VBA is built on COM; its internals are COM, your own VBA projects are embedded COM libraries. And in COM, classes cannot have parameterized constructors. Instead, we use a *factory* to create and return an initialized object. To access an external system or an object that already exists, we'll call it a *provider* instead.

If we take the action of creating and initializing an object, and turned it into a responsibility, a concern of its own – and followed the *single responsibility principle*, we could have a class that's solely responsible for exactly this, and we would call such a class a *factory*. However unless we're implementing an *abstract factory*, we rarely *need* to do this, because a *factory method* is much more convenient and just as effective.

Factory Method

This is what I consider VBA's take on parameterized object initialization: a `Create` method that is invoked off a stateless *default instance* of a class, and returns a new instance of that same class.

Without Rubberduck this can get annoying: the `VB_PredeclaredId` attribute is hidden, so to modify it you must export the class, locate and open the file with a text editor (Notepad, Notepad++) to change the `False` attribute value to `True`. Then you save and close the .cls source file and text editor, and import the modified file back into your project, and now there's a globally-scoped *default instance* of that class that you can access from anywhere by its name; the default instance of a class is always named after the class type, so the default instance of `Class1` is named `Class1`, just like the default instance of `UserForm1` is named `UserForm1`.

With Rubberduck, we simply annotate the class with `@PredeclaredId` and let the add-in do the rest.

If we treat this free global instance as "poisonous" (as we probably should: global state should generally be avoided), we avoid using it to store *instance state*, meaning we'll want to keep this default instance *stateless*. There are ways to guard against misuse, and this is as close as a VBA class gets to `Shared` behavior in VB.NET (`static` in C#), but without the compile-time assistance of a language-level keyword.

The `Create` factory method is intended to be invoked from the default instance in places where a `New` operator would otherwise be used.

I like having a factory method as the first member of the class module, perhaps as a nod to parameterized constructors.

For a `Class1` module, it could look like this:

```
Public Function Create(ByVal Args As Object) As Class1
```

However because the *default interface* of `Class1` now includes a `Create` method, the factory method would exist on the returned interface!

The solution is to return a *non-default*, explicit interface that we define in a separate class module and might call `IClass1`, where the "I" isn't really needed, but it signals that the class is an abstract interface while making it braindead-easy to name the interface: `I+ClassName` signals we're looking at an interface that's

clearly intended for *that* class, not just some random functionality the class happens to be implementing via some interface.

This interface can expose get-only properties; if we consider factory methods VBA's take on *parameterized constructors*, then an explicit interface that exposes get-only properties, if implemented correctly, would be VBA's take on *immutable objects*.

So instead of returning `Class1`, we return a new `Class1` object but the client only sees it through the lens of the `IClass1` interface, and that is *polymorphism* in a nutshell!

```
Public Function Create(ByVal Args As Object) As IClass1
```

In its simplest, parameterless form, the pattern would be implemented with a parameterless function that creates a new instance of `Class1` and returns a reference to it.

Abstract Factory

If we extracted a factory method into its own class and made it return an abstract interface, we would have an implementation for an abstract factory. The pattern then demands that we extract an interface and consume this factory class via that interface, such that the class consuming it is completely decoupled of any concrete classes and only works with abstractions. This has interesting applications: you can inject a factory that creates a completely different implementation of the output interface, and the client code would not bat an eye. This pattern is useful whenever you cannot fully initialize the returned class type at the *composition root* where you manually perform all the dependency injections and *compose* your application.

So an abstract factory is really just an interface exposing a `Create` function that takes the parameters you want to initialize the factory's *product* with, and returns an abstract interface to decouple the concrete resulting type from the calling code.

For example an `ISomethingFactory_Create` implementation would create and initialize a `New Something` instance, but would return an `ISomething`, keeping the `Something` type an internal implementation detail of the factory class.

This pattern is extremely useful with *Dependency Injection*, as it decouples not only the factory type, but also the concrete type of the factory's *product:* the created object necessarily implemented the expected interface, but how it does it is abstracted away. For example if the abstract factory yielded a `IFileSystemProvider` object, the implementation might be creating a `FileSystemObject`, but it could also be a test method injecting a *stub* that merely tracks invocations without actually hitting any actual file system; it should make no difference to the class consuming the created `IFileSystemProvider` object.

Appendices

Appendices

Index

Glossary

- Abstraction: using abstract terms to hide details from view. A variable may *abstract* an *expression* or a value; a module may *abstract* a particular functionality.
- Accessibility: determines whether an identifier can be accessed beyond the scope it's defined in, controlled by *access modifiers* such as `Private`, `Public`, and `Friend`.
- Annotation: a special comment that begins with `@`, that Rubberduck interprets as metadata.
- Argument: an *expression* that is evaluated at run-time, of which the result is passed to a procedure's parameter. Unless specified otherwise, an argument is always *positional*, and the corresponding parameter has the same position in the signature.
- Argument (named): an alternative way of passing argument expressions that uses the special `:=` operator to allow specifying arguments out of order, or to skip optional parameters in a signature.
- Attribute: refers to hidden instructions in VBA code modules that VBA interprets as metadata. Rubberduck inspections can detect attributes associated to an annotation, and synchronize their values.

- Binding: in the context of early/late binding, this term refers to the ability of the compiler to determine the address of a procedure or method. In a UI context, the term refers to the ability to link a control property's value to the value of a property of another object.

- Class: an object data type defined by a class module.
- Client: the caller of a procedure or method, or the code that consumes a given class or interface.
- Callback: a procedure that gets invoked by the run-time, like event handlers.
- COM: The *Component Object Model;* a legacy platform for the development of Windows applications, upon which VBA and VB6 are built.
- Command: an OOP design pattern that abstracts the execution of a UI command.
- Compilation: the act of translating language tokens into lower-abstraction instructions.
- Compiler: a program that parses code and turns it into executable machine instructions.
- Composition: a "has-a" relationship between two objects where one is an instance-level dependency of the other.
- Control flow: statements that control the flow of execution, including loops and conditionals, `Exit`, `GoTo`/`GoSub` and `Return` statements.

- Data type: determines the amount of memory allocated to a value. For example, a `Long` integer allocates 32 bytes.
- Dependency Injection: a technique that consists in isolating the dependencies of a class or procedure, and providing these dependencies from the client side, *injecting* them at instance level via *property injection* (the class exposes a writable property to receive that dependency) in VBA, or via *method injection* (a method receives its dependencies as parameters).

- Encapsulation: the ability of an object to keep internal state private and provide a layer of indirection that abstracts away the implementation details of that object.
- Error handler: a subroutine that runs within a procedure scope when a run-time error occurs, given an active `On Error` statement.
- Event handler: a `Sub` procedure that has a two-part identifier name consisting of the name of the

event provider and the name of the event, separate by a single underscore. Invoked by VBA when the event is *raised* by the provider.

- Expression: a very generic term that encompasses every operation that involves an operator (where the operands are also expressions themselves!), including unary operators such as the `.` (dot) *member access* operator.

- Factory: an object that is responsible for creating instances of a particular class type.
- Factory (method): a method that creates and returns a new instance of the class it is defined in, invoked from the class' *default instance*.
- Factory (abstract): an abstraction (interface) representing a factory that creates a concrete type but only exposes it as an interface; decouples the factory object from the class it creates.
- Field (instance): a module-level variable in a class module.

- Inheritance: one of the pillars of OOP, represents a "is-a" relationship between two class types, where one is *derived from* the other. For example a `ThisWorkbook` class exposes its own members but also *inherits* those of its *base* `Workbook` class. Inheritance is not supported in VBA; use *composition* instead.
- Instance: a run-time object allocated in memory, created using the `New` operator or the `CreateObject` function.
- Interface: defines [or refers to the] members exposed by an object.
- Interface (abstract): a class module whose *default interface* exposes members intended to be implemented by other class modules.
- Interface (default): the public members of a class module. Note that public fields are exposed as read/write properties.

- Inversion of Control (IoC) Container: a powerful object that can register, resolve, and release a dependency graph. Used in many languages to manage and automate the 3 stages (register, resolve, release) of DI.

- Macro: a public procedure usually without any parameters, invoked from the host application or attached to a shape or button.
- Member: any module-level declaration, including constants, variables, library imports, and procedures.
- Method: `Sub` or `Function` procedure exposed by an interface (implemented or not).

- Object: a run-time instance of a class, presenting a *default interface* defined by a class module.
- Object-oriented: a programming paradigm where objects and their interfaces form the basic building blocks of a program..

- Parameter: a local variable that is declared with a dedicated syntax in the signature of the procedure.
- Polymorphism: the ability to present the same interface for different underlying object types.
- Procedural: a programming paradigm where procedures and modules form the basis of abstraction levels.

- Refactoring: an editing operation that consists of modifying the way code works, without modifying what it does. Renaming a variable (and updating all its references) is an example of a simple refactoring.

- Scope: an execution context inside which a given identifier name has a particular specific significance. There are 3 scopes in VBA: global/project, module/instance, and local.

- **Source (version) control:** a technology that secures the history and authorship of source code and text files in a project or folder structure.
- **State (global):** values that are accessible from anywhere in the project, exposed by modules and/or referenced libraries. Includes variables in the global scope and the *instance state* of globally accessible objects, like `Excel.Application`.
- **State (instance):** the different values held by an object at run-time, whether internal/private, or publicly exposed. Instance state is always mutable (can be written to by any member of the class) in VBA, even if externally read (get) only.
- **Test (unit):** a small procedure that sets up expectations and dependencies, invokes the method under test, and compares expectations against the actual outputs; if the assertion succeeds, the test passes.
- **Token:** a grammatical "word" in a programming language, for example `Dim` or `End Sub`, but operators and identifiers are also tokens.
- **Variable:** an identifier that refers to a specific address (or range of addresses) in memory. Unless an explicit *data type* is specified, a variable declaration allocates a `Variant`

Self-published June 1, 2023, as a PDF document download in the Rubberduck Ko-fi shop for C$19.99
(50% discount for subscribers), item# RDPDFDL001.

If you did not acquire this document via a Ko-fi download,
consider making a small donation to the author at https://ko-fi.com/rubberduckvba.

Original article is available for free at https://rubberduckvba.blog/2021/05/29/rubberduck-style-guide.