Clean VBA Code pt.1: Bad Habits

Posted on October 14, 2018 by Rubberduck VBA

We know *clean code* when we see it. Clean code is a pleasure to read and maintain. Clean code makes its purpose *obvious*, and is easily extended or modified. I cannot recommend Robert C. Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* enough – to me it was an eye opener. Code examples are in Java, but the principles are mostly language-agnostic – and the realization that the vast majority of it could also be applied to VBA changed how I saw VBA code, quite radically over time.

Sometimes deeply rooted, some habits we've been carrying since forever – things we never even thought for a split-second could be second-guessed, things VBA programmers do, stem from how code was written back in the 1990's.

VBA is essentially stuck in 1998. Most of its commonly agreed-upon best practices are from another era, and while developers in every single other language moved on to more modern conventions, a lot of VBA folks are (sometimes firmly) holding on to coding practices that are pretty much universally considered harmful today: this has to be part of why so many programmers dread maintaining VBA code so much.

Is Rubberduck *enforcing* any of this?

Rubberduck **will never force you** to change your coding style. If we implemented an inspection inspired by any of these *guidelines*, it was to make it easy to identify the code that doesn't adhere to them – every inspection in Rubberduck can be disabled through inspection settings. **You own your code, you're in charge.** Rubberduck is just there to help take action if you want to, not to boss you around.

Bad Habits

In no particular order:

Systems Hungarian

If you haven't read Joel Spolsky's excellent *Making Wrong Code Look Wrong (https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/)* yet, by all means take the time *now* – it's a very, very good read, and throughout the years I must have read it a dozen times, and linked to it a hundred times.

Done and bookmarked? Ok. So now you know *Hungarian Notation* was never intended to encode data types into variable names, and that this practice only came into existence *because of an unfortunate misunderstanding*. The intent was to use short prefixes to identify the *kind of variable* (not its data type), in the context of the application. This is *Apps Hungarian* – as opposed to *Systems Hungarian*. The former is very useful and still relevant to this day, the latter is essentially useless clutter. No modern naming best-practices encourage this unfortunate prefixing scheme – it may be a hard one to unlearn, but it's worth it. Plus it spares the world from this …gem:

```
Dim oRange As Range
```

v-for-variant, i-for-integer, l-for-long (what one has to be the worst), s-for-string (or worse, "str"), o-for-object… all are useless prefixes that serve no purpose, obscure a variable's name, and that's just when they're *correct*: when they're incorrect or inconsistent, they start getting distracting and bug-inducing, on top of just being mildly annoying …or amusing. What does `strString As String` (the ultimate tautology!) tell you about the **purpose** of a parameter? That's right, nothing at all.

Disemvoweling

In BASIC 2.0 on a Commodore-64 you had a whole 2 meaningful characters to name your variables. You *could* use more, but the first 2 had to be unique.

How awesome is it that things have changed! In VBA an identifier can be up to 255 characters long. **Programming isn't about *writing* code, at least 90% of it is about *reading* code**. The handful of keystrokes you're saving are turning into tenfold the amount of time wasted investigating the meaning of these cryptic variables.

Stop stripping the vowels from variable names for no reason: they're essential to convey meaning (at least without needing to then clarify in a comment). The few spared keystrokes aren't worth all the "fun" you'll have re-reading that code in a year's time.

Wall of Declarations

I was taught to begin all procedure scopes with the declarations for all the variables in that scope, supposedly to enhance readability. For years, it seemed like a good idea – until I had to debug a 700-liner legacy procedure that started with a literal *wall* of declarations… with half of them not used anywhere, and the whole thing taking up *more* than a whole screen's height. In fact, *every single time* I answered (or commented on) a question on Stack Overflow and noticed a variable wasn't used anywhere, there was a wall of declarations at the top of the procedure.

Declare variables where you're using them. That way you'll never need to wonder if a variable is used or not, and you'll never waste considerable time constantly scrolling up & back down, then back up, then down, when debugging a large procedure.

Code that is easy to maintain, is code that is easy to modify, and thus easy to *refactor*. Having 10 lines of declarations at the top of a procedure scope isn't working in that direction: as the code changes, the maintainer will be more inclined to keep the style that's in place, i.e. to *append* to the list of declarations so as to keep all the declarations together… whereas if there's no such list in the first place, starting one will *look wrong*.

Banner Comments

Procedures should be responsible for a little as possible. *One thing*, ideally. Whenever there's a comment that looks like this in the body of a procedure:

```
'==== reticulate splines ====
```

It's a missed opportunity: the procedure *wants* that chunk of code extracted into its own `ReticulateSplines` scope, taking in parameters for whatever local variables it's using… and this ties back to the *Wall of Declarations*: if the variables are declared close to where they're first used, then extracting that chunk of code and knowing what declarations to bring over to the new scope, becomes much easier… and accidentally leaving unused variables behind is in turn much harder to do now.

Banner comments literally *scream* "I'm doing to many things!" – don't split procedures with banner comments. Write smaller procedures instead.

Snake_Case_Naming

Everywhere you look, in every standard type library you can include in a VBA project, everything uses a standard **PascalCase** naming style. By adopting a consistent `PascalCase` naming scheme, you make your code *blend in* seamlessly. But this isn't just a personal preference thing: `Snake_Case` **cannot** be consistently applied to any object-oriented code written in VBA, because you can't have a method named `Do_Something` on an interface. The compiler will simply refuse to consider `InterfaceName_Do_Something` as valid: because you used `Snake_Case` on a public member name, your code is now broken and can't be compiled anymore. And if you drop the underscores *just for interface methods*, then you're no longer using a *consistent* naming style, and that's a problem when consistency is king (and it *is*!).

(to be continued…)

Posted in tutorials, vbaTagged best-practices, tutorial, vba

Published by Rubberduck VBA

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

8 thoughts on “Clean VBA Code pt.1: Bad Habits”

1. **Christopher J. McClellan** *October 14, 2018* *Reply*

   Re: Wall of Declarations

   Steve McConnel’s “Code Complete” offers references to studies that prove reducing the scope of variables (moving the declarations to where they’re used) reduces bug rate. I highly recommend it to any developer looking to improve their code.

   1. **Lukas** *October 15, 2018* *Reply*

      So instead of puting declaration of all variables on top of the procedure/function it is better to declare them right before we use them first time?

      Exmple (please ignore variable names):

```
Public Function SHA1Hash(content As String) As String
  Dim asc As Object
  Set asc = CreateObject("System.Text.UTF8Encoding")

  Dim enc As Object
  Set enc = CreateObject("System.Security.Cryptography.SHA1CryptoServiceProvider")

  Dim bytes() As Byte
  bytes = asc.GetBytes_4(content)
  bytes = enc.ComputeHash_2((bytes))

  Dim pos As Long
  For pos = 0 To UBound(bytes)
    SHA1Hash = SHA1Hash & LCase(Right("0" & Hex(AscB(MidB(bytes, pos + 1, 1))), 2))
  Next

  Set asc = Nothing
  Set enc = Nothing
End Function
```

      view raw SHA1Hash.bas hosted with ❤ by **GitHub**

      vs

      Rubberduck VBA *October 16, 2018*

Exactly! Doesn't the first snippet feel much cleaner?

1. **RandomCoder** *November 16, 2018*

   In the specific case of VBA here, I'll add one late grain of salt: the lack of block scope makes some "declare it when you first use it" placements definitely less clear.
   Because of it, there is no lack of booleans/indices/counters first being set in a loop's body (usually nested, plus a With block thrown in for good measure) and used at a complete different nesting depth later. Boil it down to my unfamiliarity with VBA and neglect of source tools, but this led to some frustrating moments.
   Another was when I added some code long before a block would declare a variable with the same name. 'i' is a great name for an index, please don't hog it to yourself.
   So for clarity, my VBA projects declare every variable where they start mattering in terms of scope. However, I'm well aware of the banner comment part of your post and they're never too far from where they're first used, really.

   Finally, Lukas' example here is a bit too simple and short to be representative. A single nesting level (the loop doesn't add anything) on half a screen isn't enough to demonstrate the two styles' strengths and weaknesses.

   Rubberduck VBA *November 16, 2018*

   While on one hand I agree that lack of scoping levels below procedure level makes a rahter strong case for avoiding declarations inside e.g. For…Next or If…End If blocks, on the other hand I would argue that the problem is completely avoided if the loop body or conditional code is extracted into its own scope – and then… well, it *does* have its own scope… and the result is smaller, more specialized procedures that do less and have fewer reasons to fail. A procedure that needs to reuse an "i" counter is very likely doing too many things and should probably be split up.

2. **Lukas** *October 18, 2018* *Reply*

   With such small amount of code the differece is barely noticable.
   But I've reorganized much more complex code in one of my projects and right now looks cleaner and neater. And what is more important it's more readable and it's easier to understand.
   Great tip.

   Regarding the PascalCase – I like to name private methods and variables using camelCase and use PascalCase only for public functions/methods/procedures.
   What do you think about that?

   1. **peter roth** *December 30, 2018* *Reply*

      Re Lukas' use of mixed camelCase and PascalCase: my rule for the past few decades has been: camelCase for parameters, PascalCase for ALL procedures (cf Initialize and Terminate). Most of my local variables are lower case of 3 of 4 characters.

      Glad I finally found you folks!

3. **cc2a358c60** *January 14, 2019* *Reply*

   I consistently use Joel's suggestion for marking "unsafe" user input with the prefix "us" and "safe" data with the prefix "s". – Sadly, the check for Systems Hungarian now produces false positives, because, obviously, I have strings starting with "s" (for "safe", not for "string")…
   But, please, leave it on by default! I'll suffer for the greater good and disable the check. 🙂