Bullet-Proof Your Microsoft Excel VBA Project

By
J.L. Latham

Table of Contents

Questions and Comments are welcome via email to: HelpFrom@JLathamSite.com

At the time of this writing I am a member of the Microsoft Most Valuable Professional program in the Excel Group. An honor that I have been privileged to receive annually since 2005.

Other Organizations I regularly participate in:
IEEE Computing Group (Member)
Carnegie-Mellon SEI (Platinum Member)
American Society for Training and Development (Member)

WRITING BULLET-PROOF CODE

It can't be done. But making an effort is usually worth the effort, especially if you can make some of the effort habitual.

Let's face it, unless you have a dedicated team writing and executing test plans, you cannot guard against every possible data combination and you certainly cannot always predict what a user will try to do just because they have an "I wonder what…" moment.

Want proof? For over a decade I worked on a life-critical system and we put our software through rigorous testing before any release. First it went through in-house testing with test data that tested every feature at the upper/lower and mid point of boundaries, with and without valid user entries. It was tested using recorded data from every case in the past that had caused a problem. We even had a designated test team playing the "I wonder what…" game with it. Once it passed that testing and ran solidly without failure for 30 days, we sent it to two designated test sites to run for another 30 days in the real world while being watched closely. In an almost classic situation, after passing all of that testing, one version (out of the 22 releases I assisted with) failed within 24 hours of release – and **it failed at one of the two field test facilities**!

So, what are some of the things we can routinely do to make our software as resilient, robust and IAK (idiot at keyboard) proof?

1 ENSURING CONSTANT AND VARIABLE INTEGRITY

1.1 To Be Explicit or Not

Well, let's be frank about this: we are all adults (all programmers are performing an adult task and so, regardless of their physical age, we will give them adult status – and that does mean that they should act as responsible adults, i.e. no intentional malicious coding allowed). Since we are now all adults, we can be Explicit.

Figure 1 Option Explicit in Effect

Your initial view of a code module may not contain the *Option Explicit* statement at the beginning of it. It should – quite simply this is **your first step to responsible coding** through the use of accepted **Best Practices**.

Option Explicit is a directive to the compiler that says that all user defined constants and variables must be declared before actually using them in the code. The up side of using *Option Explicit* is that errors in your code due to typographic errors or reuse of a variable as the wrong type are greatly reduced and when it does happen, the problems are more easily identified. The down-side? Just that you have to take the time to go back and declare constants or variables that you find you need during code development.

To make sure that you don't forget to always use *Option Explicit*, you can tell Excel's VBE to always start new code modules with that statement. This is a 'permanent' setting and affects all projects you create in any workbook after making the setting.

Start by selecting [Tools] | Options from the VBE menu toolbar:

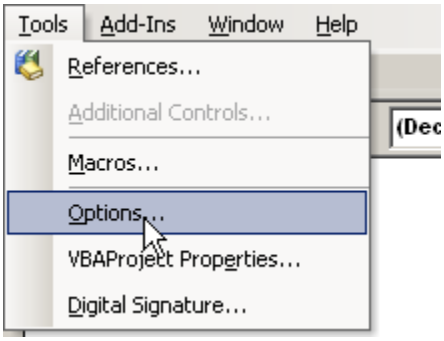
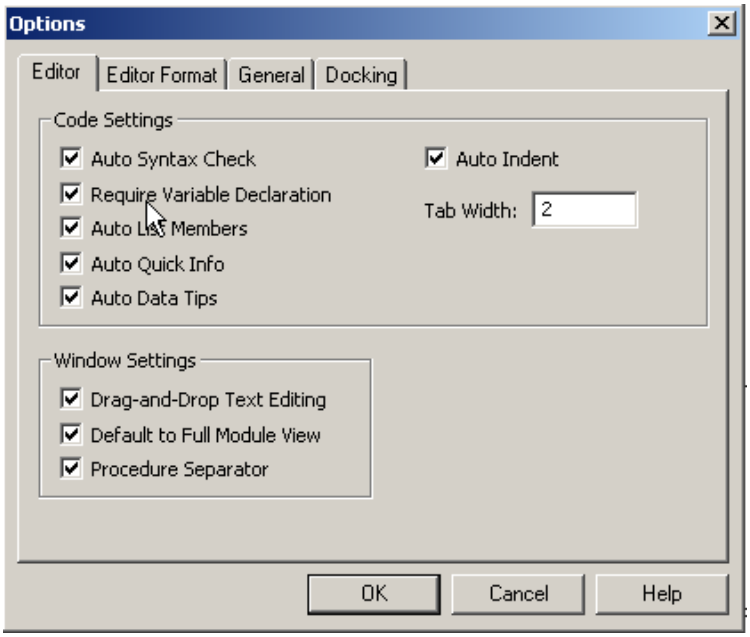


Figure 2 Setting *Option Explicit* Directive: Step 1



This is the dialog that appears once you use [Tools] | Options from the VBE menu toolbar.

Check the "Require Variable Declaration" box to set up the VBE to always place the *Option Explicit* statement at the beginning of all new code modules in the future.

Figure 3 Setting *Option Explicit* Directive: Step 2

That is all there is to it. A set it and forget it type of deal.

2 Failure Potential Cause: Worksheet Renaming or Moving

There are several ways to refer to worksheets in VBA code. The most often used method is probably by using the name of the sheet that appears on its name tab when you view it in the regular Excel window. Code might look something like this:

Worksheets("Some Sheet Name")

2.1 The Problem When Using Worksheet Tab Names

The problem with this method of referring to a worksheet is that the user *must not* change the name of the sheet. If they change the name of the sheet as they see it, then the code will fail because there is no longer a sheet with the right name in the workbook.

2.2 The Problem with Using Worksheet Index Position References

A second way of referencing a worksheet is by its position, or index, in the list of worksheets in the workbook. The position/index begins with 1 and increases as you look at the sheets from left to right in the regular Excel window. So you could reference the first (left-most) worksheet in a workbook like this:

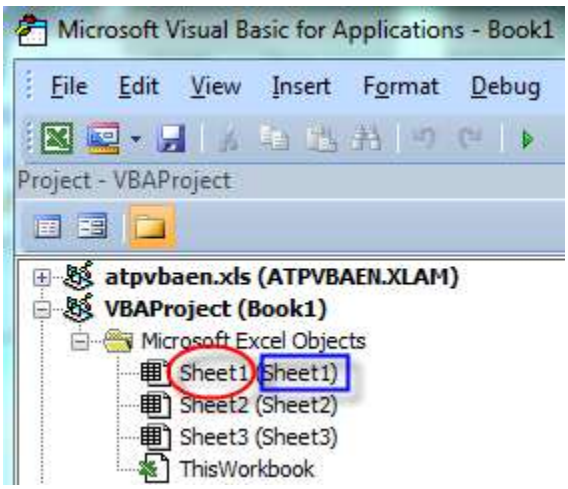
Worksheets(1)

If the user moves the worksheet, or if they insert or delete a sheet that is on the left side of the sheet in question, then the index number changes. The code probably won't fail, but it may (probably will) try to perform operations on the WRONG WORKSHEET. This in itself could be worse than simply not being able to find the right worksheet.

So what can we do to allow the end user to rename, move or even add/delete sheets to the workbook without affecting how the code works?

The answer to that is to use the worksheet's 'code name' in your code.

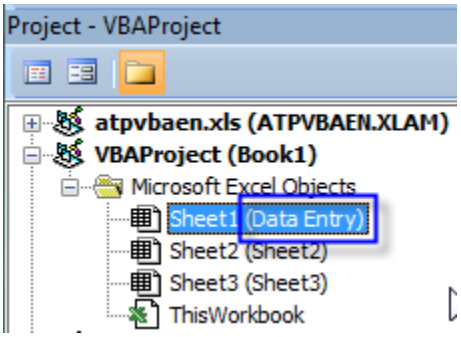
2.3 What's In a (Code) Name?



You can see both the worksheet tab names and the VBA code name in the Project window in the VBA Editor (VBE).

When you initially create a workbook and as you add sheets, both the code name and the worksheet tab name are the same.

Here the code name is shown in the red oval while the name as it appears on the worksheet's tab is shown with the **blue box**. If we change the name as it appears on the worksheet's tab, then we might see this in the window:



Here we have changed the worksheet's name on its tab to become **Data Entry**. But you will notice that the code name, Sheet1, remains unchanged.

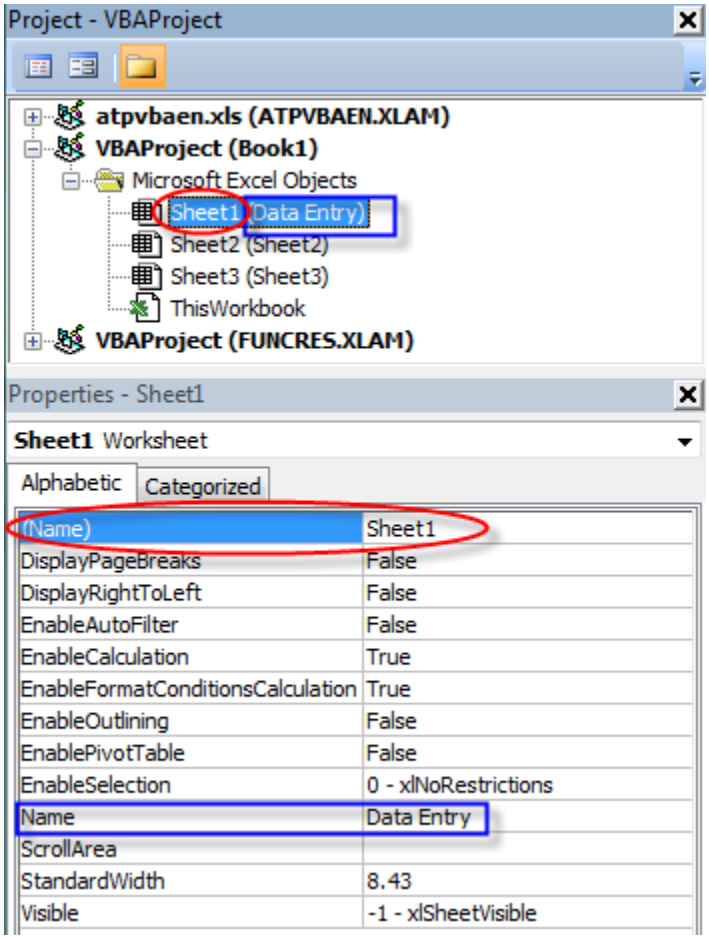
And even if we move that sheet within the workbook, the names remain the same and don't even change order in this window because they are sorted alphabetically by code name.

So, how would **we refer to a sheet in VBA code using its code name**? Actually, pretty simply:

Sheet1

would do it; notice that there are no quote marks around the name. What you must understand at this point that referring to one of the other sheets as

Worksheet("Sheet2") and as Sheet2 are two different operations. Worksheet("Sheet2") is referring to it by the name that appears on its tab, while Sheet2 is referring to it by its "internal" VBA code name.



The only way that the code name is going to changed is if someone goes into the VBE and changes the property for the sheet in the properties window.

Both names are shown in the Properties sheet when you select a sheet in the Project window. Strangely, Microsoft reverses the () on them between the two windows. The ones I have circled in red are the code name for the sheet, while the ones **boxed in blue** are the worksheet tab name.

You can actually go into either of these areas and change one or both of the sheet names. But while you can change the worksheet tab name by simply typing a new one on the sheet's tab, you cannot change the code name except through the VBE and this Properties sheet.

This means that the code name of worksheets in a workbook becomes a very safe, solid and reliable way to refer to worksheets in the workbook in your code.

Note: you cannot use code names to reference worksheets in other workbooks, and so you can't use them in general purpose macros you may have in an add-in or in your Public workbook. So, assuming Workbooks(2) is a workbook other than the one with the code in it, you cannot refer to Workbooks(2).Sheet1 from code in Workbooks(1). It just won't work and you have to fall back to worksheet names or index values and keep your fingers crossed.

If you want to take it to the next level, you can actually **change the code names** of sheets so that when you read them in your code, they help with self-documentation even more. Which is more informative to you?

Sheet1.Range("B8") = CalculateAge(DateOfBirth)

Or

PersonalInfoSheet.Range(CurrentAge) = CalculateAge(DateOfBirth)

? I vote for the second way.

In summary: using meaningful code names for sheets makes for a more solid application, less subject to failure because of end user actions (renaming, moving, adding or deleting sheets), and can add to the understanding of your code during maintenance.

3 USING CONSTANTS FOR RELIABILITY

Refer back to paragraph 1.1, To Be Explicit or Not for a refresher on Option Explicit if needed.

What else can we do to make maintenance easier while also making the code less subject to failure? The judicious use of Constants certainly comes to my mind.

When you declare a value in Excel VBA using the word ***Const*** as part of the declaration, you are telling Excel that the value is fixed and constant and may not be changed during the execution of the code.

By using constants, you can achieve several goals in your code:

#1 – when you refer to it in your code, you know its type and value.

#2 – by using meaningful names you help improve the readability of your code, which in turn makes it more manageable and easier to maintain.

#3 – by declaring certain 'features' of a workbook or worksheet as constants, you have a central location where you can change them if needed so that the code can continue to function properly even after making changes to the layout of the workbook or worksheet.

An example for #3: Suppose you have a worksheet that you store the current age of personnel in column B. You might declare a constant such as this for use in the code when you need to reference that column:

Public Const employeeAgeCol = "B" ' Column where employee age is stored on each row

Later on you need to revise the layout of the worksheet and you want to put their date of birth into column B, which of course now shoves the old column B over to become column C. You are faced with the daunting task of finding every place in your code where you may have used some code like:

PersonalInfoSheet.Range("B" & nextRow) = CalculateAge(DateOfBirth)

And first determining if the "B" is referring to the column B on the sheet you need to mess with or not, and then worrying if you have found them all or not. But what if you'd written that line, and similar ones, like this:

PersonalInfoSheet.Range(employeeAgeCol & nextRow) = CalculateAge(DateOfBirth)

Then all you have to do to fix it all everywhere in your code is to go back and redefine things at their declaration:

Public Const employeeAgeCol = "C" ' Column where employee age is stored on each row

So declaring constants judiciously and carefully choosing their 'scope' can really reduce the workload involved in maintaining the code while at the same time making it less failure prone.

Did I scare you? Don't know what "scope" means? I will try to discuss this in a way that will make sense to you. In the end, 'scope' means where in the code, or at what level in the code, can the constant or variable be referenced and used?

4 THE LEVELS OF SCOPE

There are 3 levels of scope in an Excel VBA project:

4.1 The Top Level Scope: Project Scope

Project – you can use the Public declaration qualifier to declare a constant or variable to have project-wide scope. That is, it can be seen and used in all Subs and Functions in all code modules throughout the project. These are useful when you have a particular value that may need to be used in more than just one area (module) of your project. These constants and variables must be declared in the declarations section of a regular code module in much this fashion:

Public Const Pi = 3.14159 ' limit Pi to 5 decimal places when used in this application.

Public Radius As Double ' declares a public variable for use in the application.

4.2 The Mid Level Scope: Module

Module – you can declare constants and variables to be available for use by all Subs and Functions within a specific module (the one they are declared in), but that are not used by modules, subs or functions other than the one they were declared in. So perhaps we only need to use Pi and Radius in one module; then we could have declared them without using the Public qualifier like this:

Const Pi = 3.14159 ' limit Pi to 5 decimal places when used in this module.

Dim Radius As Double ' declare a double-precision variable for use by any Sub/Function in this module only.

4.3 The Low Level Scope: Local to a Sub or Function

At the Sub or Function level. You can declare both constants and variables for use only in a specific Sub or Function. They cannot be seen or heard or felt anywhere else in the entire project. Suppose we had a Function that calculates the area of a circle when you pass it a diameter, but you prefer to use the old fashioned Pie Are Square formula. It might look like this:

```
Function AreaOfCircle(diameter As Double) As Double
  Const Pi = 3.14159 ' only seen and usable in this function only exists when using this Function
  Dim Radius As Double ' erased once we leave this function

  Radius = diameter / 2 ' so we can use the =Pi * r^2 formula to calculate area
  AreaOfCircle = Pi * Radius^2
End Function
```

So where is this magic declarations section of a code module? First, consider a regular code module. If you've followed my advice from my Introduction to VBA for Excel, all of your code modules will have this one-liner at the very beginning of them:

Option Explicit ' requires that all constants/variables be declared before actual use. See paragraph 1.1 for a refresher on setting this up.

Everything after that, but before any declaration of a Sub or Function is part of the *declaration* section of the code module.

The effects of different levels of scope is demonstrated in the associated workbook, SlicesOfPi.xlsm. In it there are 3 code modules.

In Module1, a Public Const Pi = 3.14159 statement is in the declarations section. There is also a Sub that simply prints the value of Pi as it sees it in a message box. The 'button' 1) on Sheet 1 of the workbook demonstrates this.

In Module3 there are no declarations, just a piece of code that again prints the value of Pi as it sees it. It will also report Pi = 3.14159 because it is using the value of Pi from the Public declaration in Module1. Button 2) on Sheet 1 demonstrates this for you.

In Module 2 we really confuse the issue:

First, there is a variable named Pi declared in its declarations section, but no value is assigned to it. If you click button 3) on Sheet 1, you will see that Pi = 0 in the message box because it has ignored the Public Const declaration of Pi in favor of the "closer" declaration of Pi as a variable with as yet undefined value.

Second, in Module 2 there is a Sub that has its own declaration of Pi set = 3. And if you click on button 4) on Sheet 1 you will see that it reports that Pi = 3. In this case, it has ignored both the nearby variable Pi declared in the same module and the Public Pi declared in Module1 in favor of the much closer declaration of Pi in the Sub itself.

Generally it is a ***very bad idea*** to use the same name for a constant or variable with different values at different scope levels. You are just asking for a disaster to take place sooner or later. But I did this intentionally in this exercise to try to help you see how scope works.

If you need more help understanding scope, I recommend that you do a search on the internet for the phrase "VBA Scope". You will get several hits and you can read up on it from different authors and hopefully one of them will be able to explain it to you in a way you can understand. A good starting point is probably Chip Pearson's take on it at http://www.cpearson.com/excel/scope.aspx

OK with my discussion of scope out of the way, back to the subject of **Using Constants for Reliability**.

I personally like to have a single code module to contain all of my Public declarations, both constants and variables. This gives me a single place to go to look for anything that might need to be changed at a later time in response to either a worksheet layout or change in processing logic involving the use of values. Within that module I try to organize the grouping by either function or by what sheets may be using them.

For example, suppose I have a sheet where the first row has labels in it and the data entries start on row 2 plus there are some specific columns where data we need to work with are located. I might have a section in the module like this:

```
'These Constants and variables are used to process information on the [DataEntry] sheet

Public Const de_FirstDataRow = 2 ' first row with a data entry in it
Public Const de_AgeCol = "B" ' column that contains ages
Public Const de_RetDateCol = "F" ' column that contains retirement dates
```

Perhaps we also need to use and be able to change the age at which someone becomes eligible to retire and that might be used several places so I might also declare this in the same module:

```
Public Const minimumRetirementAge = 65
```

That's kind of about all I have to say about this for now. The advantages to using Public Const definitions are that it makes code management easier, makes adapting the code to changes in the layout and content of worksheets easy and (almost) foolproof.

5 THE BIGGEST CONSTANT IN THE UNIVERSE IS VARIABLE

What is a variable and why use them? A variable is a named 'thing' that can be reused to hold different similar values during your processing. For an example, perhaps you have a variable you have defined as:

```
Dim anyAge As Integer
```

anyAge can now hold any whole number from -32,768 up to +32,767, and therefore it's suitable for holding people's ages.

Because we have declared it as an Integer, we can't assign a non-integer piece of information to it such as a name, or really large number such as 1 million.

But we can CHANGE its value throughout our processing – at one point it might hold someone's age that is 44 years old, at another point or time it might hold someone's age who is 3 years old.

Because we have given the variable what should be a meaningful name, anyAge, we should try our very best in our code to make sure that we only use it to hold ages, not row numbers, column numbers or bank balances. Declare other variables for those other values.

5.1 Naming Variables (and Constants)

These recommendations apply to both Variables and Constants. While I will reference them as variables, please remember that I am speaking of both Variables and Constants.

DO NOT give variables single letter names! Names like A or X or I are pretty useless for imparting any kind of information while reading the code.

I kind of 'slide' to some degree in my coding in that I may use variable names like LC, SLC, RLC and CLC – but my personal coding convention lets me know that "LC" means Loop Counter. So SLC might be a loop counter for working through a list of worksheets, and RLC may be a loop counter for working through a range of rows on a worksheet. While these are a bit cryptic to read out of context; in context they are very clear (at least to me).

Make your variable names easy to read. Variable and constant names must begin with a letter of the alphabet and they can't hold certain special characters. One of those special characters is the

space character. This means you cannot have a name like 'My Variable'. You could have one named MyVariable. You may say that it might become difficult to read a name like rowpointer and make sense of it quickly. There are a couple of solutions to this problem. One is to use what is often called camel-back naming and it ends up with names that look like:

MyRowPointerForArrayTimeline
or another way
myRowPointerForArrayTimeline
if you just like to start with lowercase to let yourself and others know that this is a variable that you gave a name to.

One character that is allowed as part of names is the underscore character, so it can easily be used to distinguish words in a name, like myRowPointerFor_A_Loop.

Some people like to start the name of their variables with a type indicator, and that's not a bad idea, but it is not mandatory. Some examples might be:

intAnyAge or iAnyAge- an integer value
bolUserChoice or bUserChoice – a Boolean variable (True or False)
lngRowPointer or lRowPointer – a Long value
sSomeName or strSomeName or txtSomeName – a string (text) value.

One obvious drawback to really long names is that it takes more type and effort to type them repeatedly in your code. So pick a reasonable compromise between length and understanding.

Be consistent in the way you create names. Always using camel-back style or always (or not) providing a type indicator as part of the name will help the readability of your code in the long run.

5.2 Use Unique Names

Do not use the same names at different levels of scope. This way lays madness! You don't want to have a variable (or constant) at different levels of scope all having the same name. When you reference something named myDef anywhere, you want to be sure of what scope it has which in turn lets you know what its value should be, or at least how to easily find out what that value should be.

I will step out on a limb a little and say that it is acceptable to use the same name for variables *at the same level of scope*. As an example, it is not unusual for you to need to determine the last used row on a worksheet for various reasons. And there is nothing wrong with having a variable named lastUsedRow in each and every Sub or Function throughout your project. Each of those would have local scope and be totally independent of the others. No chance of one of their values "leaking" over into another Sub/Function to mess up your day.

5.3 Don't Reuse Variables

This is not the same as I explained about the anyAge variable earlier. In that discussion, anyAge could hold any of several people's age depending on when and where we were in the process. What I mean by don't reuse variables is not to use them to hold values of different purpose. That is, don't use anyAge to hold row numbers, column numbers, offset values or anything other than people's ages.

So how many variables should you have? As many as you need is pretty much the answer. But that is not to say that you need a separate one for every possible variable that may come up during the processing. Sometimes you need "working" variables to temporarily store information that may be manipulated or processed through a series of steps that ends up being assigned as the final result to yet another variable with a more meaningful name. I usually start out the name of these really variable variables with tmp (for temporary) or wrk (for working).

5.4 How Do Names Improve Bullet-Proofing?

Use of good naming convention helps bullet proof your project by providing distinct places to put your information so that when you go to use it in your code you get what you expect, not something seemingly at random.

It also helps make your code more understandable when you do have to return to it for modifications, upgrades or plain old-fashioned bug fixes.

6 COMMENTS AID BULLET-PROOFING

Since comments are non-executable statements in your code, you might wonder how they could play into making it more bullet-proof.

Unless you work with a specific application on an almost daily basis, you are going to forget how it works! But the liberal use of comments can make it much easier to come back up to speed on it when the need arises. This means you can regain understanding of the process flow and actions quickly, and without that understanding it is really impossible to 'fix' or add to something without the risk of unwanted side effects.

Be liberal with your comments, but don't throw them in just because I said lots of comments were going to cure all your ills. Make the comments meaningful and explanatory. A comment that reads ***'add 1 to pointer XRay*** is probably not very meaningful since the line of code it refers to probably looks like **XRay = XRay +1** which says exactly the same thing! Explain WHY you are adding 1 to that pointer, like ***'update pointer to the next record in array XRay so we can examine the next patient's name***. Of course, that assumes that this is actually why that pointer is being incremented.

The only thing better than well documented code is to actually have a "Programmer's Guide" that explains the processing step by step for future reference.

7 BUILD AND REUSE REUSABLE CODE

It is entirely possible to write just about any process in single Sub or Function. But if there is a lot of work to be done, especially work of different general purpose, it is probably easier to deal with if you break it up into pieces.

Some of those pieces will end up being reusable. Once you have a section of code working properly, you've got good error handling built into it, and it is something seemingly solid and reliable, then it becomes a candidate for inclusion to your library of processes/functions. In other words, it becomes a piece of reusable code.

Some typical candidates for writing as separate Subs/Functions to be reused later might be:

Routine to have the user select and open a new workbook

Routine to save a particular workbook with specific conditions (save changes, or not)

Work through the files in a folder and return a list of all of them of a particular type in the folder.

Perform a frequently needed action such as determining the last row/column/cell used on a worksheet.

It should be obvious how this type of thing helps build bullet-proof code: you are putting together a new application using pieces that are presumably already bullet-proof and so they will allow you to focus on bullet-proofing the new code for the project.

8 TESTING

This may be one of the most neglected areas in most small team or home projects. But it really is crucial to the success of things at the end.

You should test as much as possible during the development of a process just to make sure that the new code works as you are putting it together.

At the end of things, you should have a pretty comprehensive system test to run through to make sure that all the pieces fit together and provide valid, reliable results.

I'll just say this about testing and then I'll wind things up:

Test with values that are:

- Within the valid limits: at the low end, in the middle, and at the high end. Example – you have a routine that is supposed to accept values from 1 to 100 inclusive. You would at least want to test with values of 1, 50 and 100 to make sure that all of them were accepted.
- Outside of valid limits: for the above example, test with 0 and 101 and perhaps -10 and 1000 and verify that those values are rejected without causing program failure.

9 VERSION (CONFIGURATION) CONTROL

There are tools available to professional developers that provide version/configuration control of applications and the code within them. Most of these are more expensive and complex than the typical home user is going to want to deal with. So we're not even going to talk about them! But I am going to talk a little about developing your own "home grown" version control system – a lot of which depends on your own self-discipline.

A lot of applications you put together for yourself may not even warrant configuration control since you may be making all of your changes to the one and only copy of the file. More about one-and-only later. If you're doing this, then the version in use is always the current version.

But in many projects you find the need to add more features, revise the layouts of worksheets, add new worksheets and make other changes that affect the capabilities of the workbook and how it interacts with other files. For these workbooks, especially if they are going to be distributed to others for their use, you very much want some version control process in place.

9.1 Version Idenfication

The first thing you need to do is decide how you are going to identify individual versions of your workbook. You could simply number them beginning with 1 and continuing from there. But that doesn't give you a lot of information about why they were changed or what drove the change.

We see a lot of software that has 3 number groups in a version identifier, like 1.2.3 or 2.4.207. I use a setup like that myself and here's how I use the different pieces of the numbers to help give me a little information about a version. Let's say I have just given a workbook that I had versioned as 3.2.0 a new number: 3.2.1. What does it all mean (to me)?

The 3 is the major series number – that only changes when something really major happens such as adding or deleting an entire feature to/from the application. Such a change would make it pretty much incompatible for use with previous versions.

The 2 is a major revision to the series number. It would change when a bug was fixed, or a change in the way some process was performed (a change in program logic).

The 1 out there in the 3rd place indicates something approaching an editorial change. It might be just that: a correction of spelling on a button label or repositioning some informational cells on a sheet. That type of thing.

There are other identifying schemes, I've presented one that works for me. That's the key: that it works for you to identify various versions of a project.

What to do with that identification? If you have VBA code in your workbook, you could add your own [About…] button and when clicked it would show the version and perhaps some contact information so users can get in touch if stuff happens while using it. For that you would probably want a Public Const set up in the code to hold the version ID for this use.

You could also consider adding a [History] sheet to the workbook to record at least some highlights of what changed in each version and why. You could make that sheet hidden if you like, but it would serve as an informal record of change for your reference if you aren't keeping any other documentation on the whole package.

9.2 Configuration Control

It's best not to have all versions of your workbook in a single folder. For any of my projects I have 3 basic folders under a major folder (which contains nothing but the 3 subfolders).

1) Released Versions. This folder contains the final copy of each version with its version identifier as part of the filename.
2) Current Version. I keep a copy of the latest released version in this folder – actually 2 or 3 copies!
    a. One copy as it is actually distributed. This copy normally does not have the version identification as part of the name. The reason for this is that many people may set up shortcuts to the workbook, and by just using the basic name of a workbook, people can simply save over their older version and their shortcuts continue to work.
    b. One copy with the version number – more as a reminder to myself of the current version number than anything else, even though there would be a second copy of this with the same name in the Released Versions folder.
    c. If I have locked the VBAProject to prevent tampering by the end user(s), I will also save a copy that does not have the VBAProject locked just in case I forget or lose the password to the project. That's a tough password to crack, so it's easier and quicker in the long run to keep an unlocked copy around "just in case".
3) Next Version: This is a folder where I put a copy of the latest version, give it a new version number and work on changes that will eventually become the next version.

Remember, when you distribute the workbook, it's probably best for the end users to always distribute it with a name that does not contain a version number.

If you're wondering how this helps bullet-proof things, just think of it as a way of not accidentally re-using a package that is less bullet-proof as you go forward in the creation of it.

10 BACKUP STRATEGY

I mentioned "one-and-only" copy earlier. You NEVER want to be in the position of having just one copy of pretty much anything on your computer. Motherboards short out, disk drives fail and people sometimes delete the wrong file. You want to protect yourself from these kinds of situations so that you don't have to design the whole masterpiece from a blank canvas all over again.

A good backup strategy involves using an automated backup tool to at least copy critical files to a separate hard drive.

A very good backup strategy involves using an automated backup tool to at least copy critical files to a hard drive on a separate device (another computer or large storage device).

A really strong backup strategy involves adding the ability to have off-site storage of your critical files in addition to the features of a "very good" backup strategy. This helps you with recovery from really big disasters like flood, fire or tornado.

There are very good, and relatively inexpensive automated tools available. One of my favorites is SecondCopy from Centered Systems (http://www.secondcopy.com/). I've been recommending this product to many of my clients and friends for over 10 years. It works very well, can backup between multiple systems, is very reasonably priced, and it is easy to setup and use and it is reliable! At $29.95 (as of today) it is one of the best buys around. I must mention their great licensing arrangement: you can get away with a single copy installed on a single system to backup multiple systems! They license by the number of installations, not on the number of computers involved. So if you had a home network with several computer on it, you could set it up on one computer, and if you can see another computer/drive from that one, you can include that other computer as either the source (to be backed up) or destination (where your backups are stored) for any backup profile you create with it.

Another one I have experience with is BackIt-Up from Nero systems. It's available stand-alone for about $29.99 or as part of some of their multi-application packages/suites. Once again it's simple to setup and use and inexpensive to acquire.

Some external hard drives also come with similar software. Western Digital Passport drives come to mind for me. When you attach one to a computer via USB, it installs backup software to automatically backup from your internal hard drive(s) to the Passport drive. Something to consider if you only have one computer and no other place to backup your files to. And if you buy 2 of the same model, you can even swap them out and take one of them to your offsite location (perhaps to your workplace).

What about cloud storage? It's an option. I personally consider it a better-than-nothing option. Basically anytime you store your data on someone else's system, you make it available to them to do with as they please. Plus you're dependent on a functioning internet connection to both backup data and to restore it. Finally, many of the on-line storage/backup services DO NOT

provide any type of security for your files – a recent survey showed that about 75% of them considered security of the files to be the customer's responsibility. Since you probably aren't allowed to install any operational software on their system, this would mean at the very least that you encrypt all files placed on their servers to protect your personal and intellectual property and your privacy.

11 SUMMARY

I have presented some processes and concepts that should assist you in putting together solid working applications. I've put them together based on just over 30 years of actually programming – everything from life-critical systems to fairly trivial applications that perhaps only change the color of certain cells in Excel based on user actions. This is NOT to say that these are the end-all of such a list. There are other things that go into writing solid, reliable code – not the least of which is an understanding of:

- The Problem – unless you actually understand the problem, it is impossible to provide a solution to the problem.
- The planned procedures needed to solve the problem -- this is also an often neglected part of programming. Except for small projects, every project should involve some planning on your part before writing the first line of code. THINK about what you need to do to get the problem solved.
- The Language being used to solve the problem

and

- Testing – Testing – Testing (and testing again)