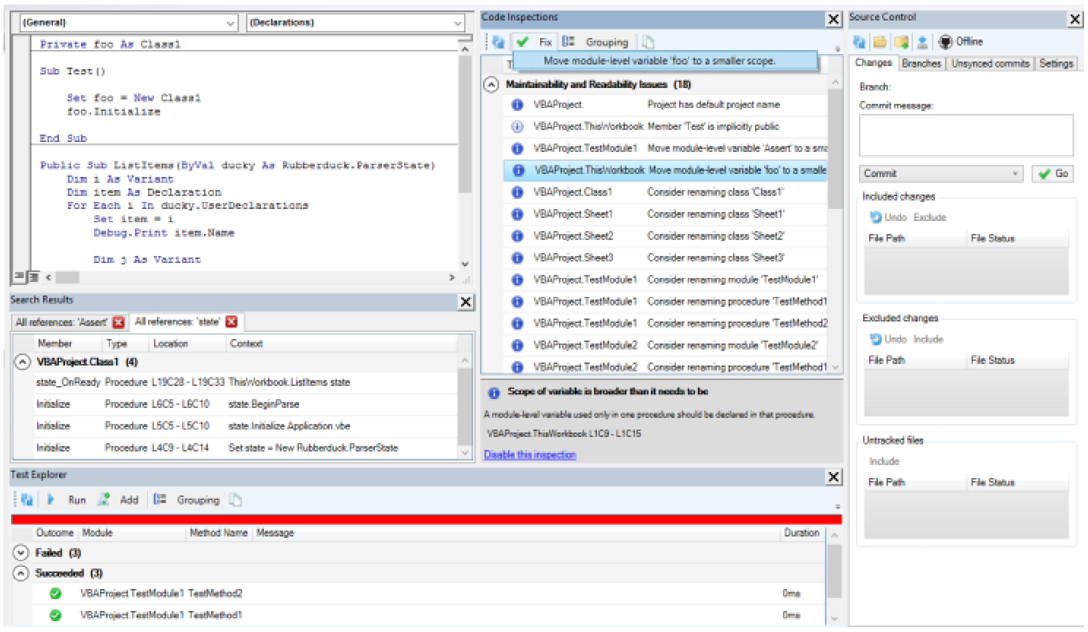
VBA Rubberducking (Part 2)

Posted on May 14, 2016May 15, 2016 by Rubberduck VBA

This post is the second in a series of post that walk you through the various features of the Rubberduck open-source VBE add-in. The first post (https://rubberduckvba.wordpress.com/2016/05/04/vba-rubberducking-part-1/) was about the **navigation** features.

Code Inspections


(General) (Declarations)

```
Private foo As Class1

Sub Test()

    Set foo = New Class1
    foo.Initialize

End Sub

Public Sub ListItems(ByVal ducky As Rubberduck.ParserState)
    Dim i As Variant
    Dim item As Declaration
    For Each i In ducky.UserDeclarations
        Set item = i
        Debug.Print item.Name

        Dim j As Variant
```

Code Inspections

Move module-level variable 'foo' to a smaller scope.

Maintainability and Readability Issues (18)

●	VBAProject.	Project has default project name
●	VBAProject.ThisWorkbook	Member 'Test' is implicitly public
●	VBAProject.TestModule1	Move module-level variable 'Assert' to a sma
●	VBAProject.ThisWorkbook	Move module-level variable 'foo' to a smalle
●	VBAProject.Class1	Consider renaming class 'Class1'
●	VBAProject.Sheet1	Consider renaming class 'Sheet1'
●	VBAProject.Sheet2	Consider renaming class 'Sheet2'
●	VBAProject.Sheet3	Consider renaming class 'Sheet3'
●	VBAProject.TestModule1	Consider renaming module 'TestModule1'
●	VBAProject.TestModule1	Consider renaming procedure 'TestMethod1'
●	VBAProject.TestModule1	Consider renaming procedure 'TestMethod2'
●	VBAProject.TestModule2	Consider renaming module 'TestModule2'
●	VBAProject.TestModule2	Consider renaming procedure 'TestMethod1'

Scope of variable is broader than it needs to be

A module-level variable used only in one procedure should be declared in that procedure.

VBAProject.ThisWorkbook L1C9 - L1C15

Disable this inspection

Search Results

All references: 'Assert' | All references: 'state'

Member	Type	Location	Context
VBAProject.Class1 (4)			
state_OnReady	Procedure	L19C28 - L19C33	ThisWorkbook.ListItems state
Initialize	Procedure	L6C5 - L6C10	state.BeginParse
Initialize	Procedure	L5C5 - L5C10	state.Initialize Application.vbe
Initialize	Procedure	L4C9 - L4C14	Set state = New Rubberduck.ParserState

Test Explorer

Outcome	Module	Method Name	Message	Duration
Failed (3)				
Succeeded (3)				
✓	VBAProject.TestModule1	TestMethod2		0ms
✓	VBAProject.TestModule1	TestMethod1		0ms

Back when the project started, when we started realizing what it meant to *parse* VBA code, we knew we were going to use that information to tell our users when we're seeing anything from *possibly iffy* to *this would be a bug* in their code.

The first one to be implemented was **OptionExplicitInspection**. The way Rubberduck works, a variable that doesn't resolve to a known declaration simply doesn't exist. Rubberduck is designed around the fact that it's working against code that VBA compiles; is also needs to assume you're working with code that declares its variables.

Without 'Option Explicit' on, Rubberduck code inspections can yield false positives.

Because it's best-practice to always declare your variables, and because the rest of Rubberduck won't work as well as it should if you're using undeclared variables, this inspection defaults to **Error** severity level.

OptionExplicitInspection was just the beginning. As of this writing, we have implementations for 35 inspections, most with one or more one-click *quick-fixes*.

35 inspections?

And there's a couple more left to implement, too. A lot of inspections rely on successful parsing and processing of the entire project and its references; if there's a parsing error, then Rubberduck will not produce new inspection results. When parsing succeeds, inspections run automatically and the "status bar" indicates **Ready** when it's completed.

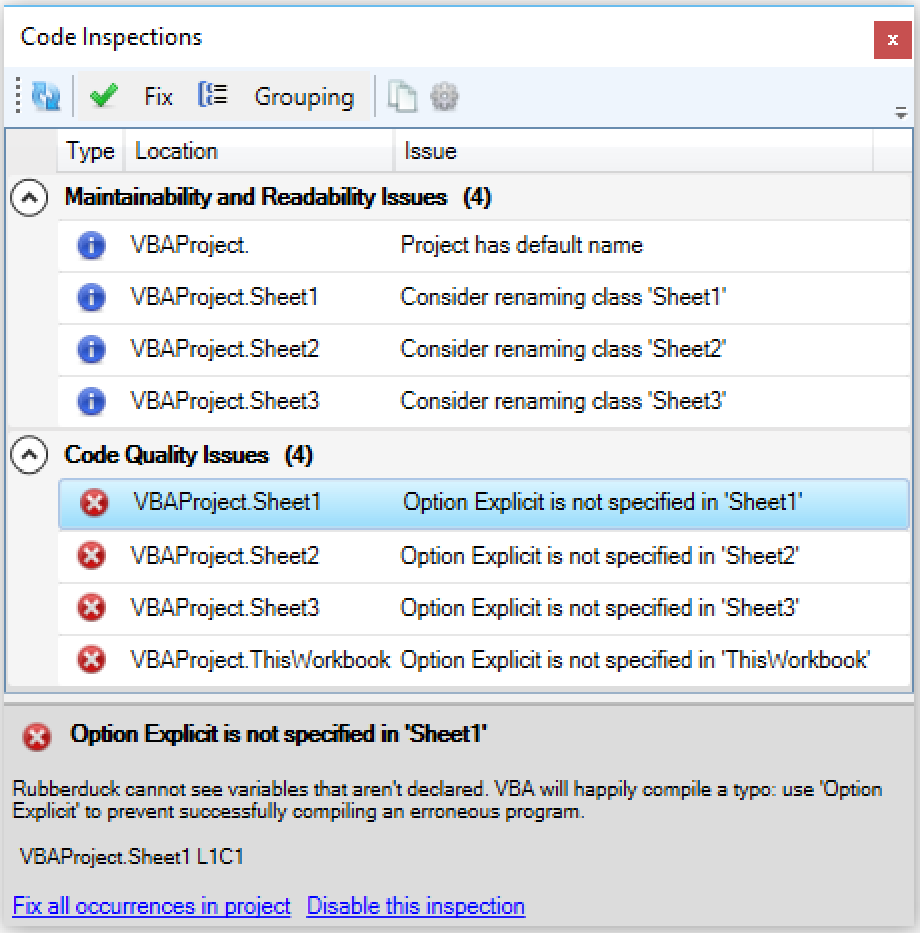
1. **AssignedByValParameterInspection** looks for parameters passed by value and assigned a new value, suggesting to either extract a local variable, or pass it by reference if the assigned value is intended to be returned to the calling code.
2. **ConstantNotUsedInspection** looks for constant declarations that are never referenced. Quick-fix is to remove the unused declaration.
3. **DefaultProjectNameInspection** looks for unnamed projects ("VBAProject"), and suggests to refactor/rename it. If you're using source control, you'll want to name your project, so we made an inspection for it.
4. **EmptyStringLiteralInspection** finds "" empty strings and suggests replacing with **vbNullString** constants.
5. **EncapsulatePublicFieldInspection** looks for public fields and suggests making it private and expose it as a property.
6. **FunctionReturnValueNotUsedInspection** locates functions whose result is returned, with none of the call sites doing anything with it. The function is used as a procedure, and Rubberduck suggests implementing it as such.
7. **IdentifierNotAssignedInspection** reports variables that are declared, but never assigned.
8. **ImplicitActiveSheetReferenceInspection** is Excel-specific, but it warns about code that implicitly refers to the active sheet.
9. **ImplicitActiveWorkbookReferenceInspection** is also Excel-specific, warns about code that implicitly refers to the active workbook.
10. **ImplicitByRefParameterInspection** parameters are passed by reference by default; a quick-fix makes the parameters be explicit about it.
11. **ImplicitPublicMemberInspection** members of a module are public by default. Quick-fix makes the member explicitly public.
12. **ImplicitVariantReturnTypeInspection** a function or property getter's signature doesn't specify a return type; Rubberduck makes it return a explicit Variant.
13. **MoveFieldCloserToUsageInspection** locates module-level variables that are only used in one procedure, i.e. its accessibility could be narrowed to a smaller scope.
14. **MultilineParameterInspection** finds parameters in signatures, that are declared across two or more lines (using line continuations), which hurts readability.
15. **MultipleDeclarationsInspection** finds instructions containing multiple declarations, and suggests breaking it down into multiple lines. This goes hand-in-hand with declaring variables as close as possible to their usage.
16. **MultipleFolderAnnotationsInspection** warns when Rubberduck sees more than one single **@Folder** annotation in a module; only the first annotation is taken into account.

17. **NonReturningFunctionInspection** tells you when a function (or property getter) isn't assigned a return value, which is, in all likelihood, a bug in the VBA code.
18. **ObjectVariableNotSetInspection** tells you when a variable that is known to be an object type, is assigned without the **Set** keyword – this *is* a bug in the VBA code, and fires a runtime error 91 "Object or With block variable not set".
19. **ObsoleteCallStatementInspection** locates usages of the **Call** keyword, which is never required. Modern form of VB code uses the *implicit* call syntax.
20. **ObsoleteCommentSyntaxInspection** locates usages of the **Rem** keyword, a dinosaurian syntax for writing comments. Modern form of VB code uses a single quote to denote a comment.
21. **ObsoleteGlobalInspection** locates usages of the **Global** keyword, which is deprecated by the **Public** access modifier. **Global** cannot compile when used in a class module.
22. **ObsoleteLetStatementInspection** locates usages of the **Let** keyword, which is required in the ancient syntax for value assignments.
23. **ObsoleteTypeHintInspection** locates usages of *type hints* in declarations and identifier references, suggesting to replace them with an explicit value type.
24. **OptionBaseInspection** warns when a module uses **Option Base 1**, which can easily lead to *off-by-one* bugs, if you're not careful.
25. **OptionExplicitInspection** warns when a module does not set **Option Explicit**, which can lead to VBA happily compiling code that uses undeclared variables, that are undeclared because there's a typo in the assignment instruction. Always use Option Explicit.
26. **ParameterCanBeByValInspection** tells you when a parameter is passed **ByRef** (implicitly or explicitly), but never assigned in the body of the member – meaning there's no reason not to pass the parameter by value.
27. **ParameterNotUsedInspection** tells you when a parameter can be safely removed from a signature.
28. **ProcedureCanBeWrittenAsFunctionInspection** locates procedures that assign a single **ByRef** parameter (i.e. treating it as a return value), that would be better off written as a function.
29. **ProcedureNotUsedInspection** locates procedures that aren't called anywhere in user code. Use an **@Ignore** annotation to remove false positives such as public procedures and functions called by Excel worksheets and controls.
30. **SelfAssignedDeclarationInspection** finds local object variables declared **As New**, which (it's little known) affects the object's lifetime and can lead to surprising/unexpected behavior, and bugs.
31. **UnassignedVariableUsageInspection** locates usages of variables that are referred to *before* being assigned a value, which is usually a bug.
32. **UntypedFunctionUsageInspection** recommends using **String**-returning functions available, instead of the **Variant**-returning ones (e.g. **Mid$** vs. **Mid**).
33. **UseMeaningfulNamesInspection** finds identifiers with less than 3 characters, without vowels, or post-fixed with a number – and suggests renaming them. Inspection settings will eventually allow "white-listing" common names.
34. **VariableNotAssignedInspection** locates variables that are never assigned a value (or reference), which can be a bug.
35. **VariableNotUsedInspection** locates variables that might be assigned a value, but are never referred to and could be safely removed.
36. **VariableTypeNotDeclaredInspection** finds variable declarations that don't explicitly specify a type, making the variable implicitly **Variant**.
37. **WriteOnlyPropertyInspection** finds properties that expose a setter (Property Let or Property Set), but no getter. This is usually a design flaw.

Oops, looks like I miscounted them… and there are even more coming up, including host-specific ones that only run when the VBE is hosted in Excel, or Access, or whatever.

The *Inspection Results* toolwindow

If you bring up the VBE in a brand new Excel workbook, and then bring up the *inspection results* toolwindow (Ctrl+Shift+I by default) you could be looking at something like this:


Code Inspections

Type	Location	Issue
Maintainability and Readability Issues (4)		
ℹ	VBAProject.	Project has default name
ℹ	VBAProject.Sheet1	Consider renaming class 'Sheet1'
ℹ	VBAProject.Sheet2	Consider renaming class 'Sheet2'
ℹ	VBAProject.Sheet3	Consider renaming class 'Sheet3'
Code Quality Issues (4)		
✖	VBAProject.Sheet1	Option Explicit is not specified in 'Sheet1'
✖	VBAProject.Sheet2	Option Explicit is not specified in 'Sheet2'
✖	VBAProject.Sheet3	Option Explicit is not specified in 'Sheet3'
✖	VBAProject.ThisWorkbook	Option Explicit is not specified in 'ThisWorkbook'

Option Explicit is not specified in 'Sheet1'

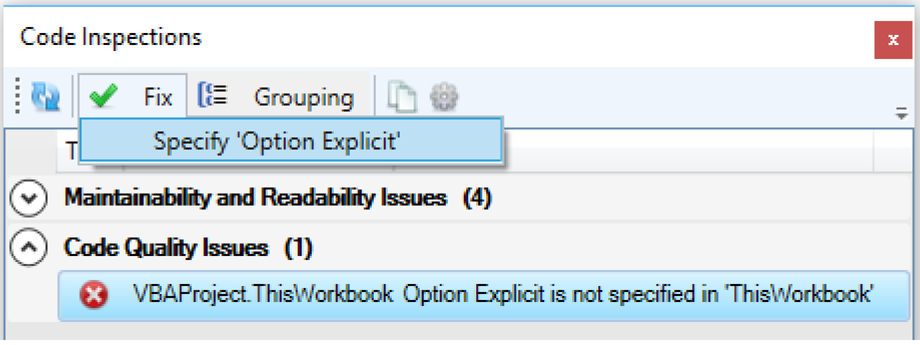
Rubberduck cannot see variables that aren't declared. VBA will happily compile a typo: use 'Option Explicit' to prevent successfully compiling an erroneous program.

VBAProject.Sheet1 L1C1

Fix all occurrences in project   Disable this inspection

Most inspections provide one or more "quick-fixes", and sometimes a quick-fix can be applied to all inspection results at once, within a module, or even within a project. In this case **Option Explicit** can be automatically added to all modules that don't have it in the project, using the blue **Fix all occurrences in project** link at the bottom.

Or, use the **Fix** drop-down menu in the top toolbar to apply a quick-fix to the selected inspection result:

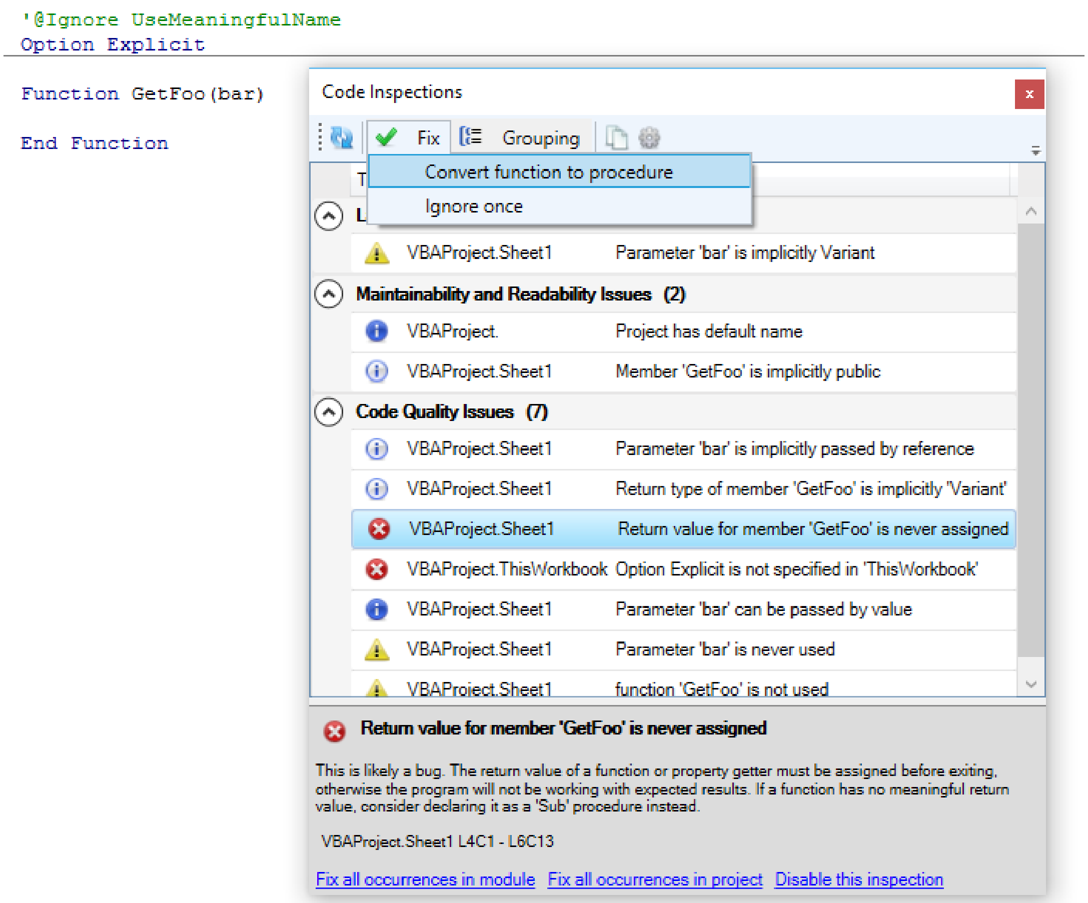

Code Inspections

Maintainability and Readability Issues (4)

Code Quality Issues (1)

✖ VBAProject.ThisWorkbook Option Explicit is not specified in 'ThisWorkbook'

Each inspection has its own set of “quick-fixes” in the **Fix** menu. A common one is **Ignore once**; it inserts an **@Ignore** annotation that instructs the specified inspection to skip a declaration or identifier reference..


Code Inspections
Fix
Grouping
Convert function to procedure
Ignore once
Fix all occurrences in module
Fix all occurrences in project
Disable this inspection

```
'@Ignore UseMeaningfulName
Option Explicit

Function GetFoo(bar)

End Function
```

⚠	VBAProject.Sheet1	Parameter 'bar' is implicitly Variant
Maintainability and Readability Issues (2)		
●	VBAProject.	Project has default name
●	VBAProject.Sheet1	Member 'GetFoo' is implicitly public
Code Quality Issues (7)		
●	VBAProject.Sheet1	Parameter 'bar' is implicitly passed by reference
●	VBAProject.Sheet1	Return type of member 'GetFoo' is implicitly 'Variant'
✖	VBAProject.Sheet1	Return value for member 'GetFoo' is never assigned
✖	VBAProject.ThisWorkbook	Option Explicit is not specified in 'ThisWorkbook'
●	VBAProject.Sheet1	Parameter 'bar' can be passed by value
⚠	VBAProject.Sheet1	Parameter 'bar' is never used
⚠	VBAProject.Sheet1	function 'GetFoo' is not used

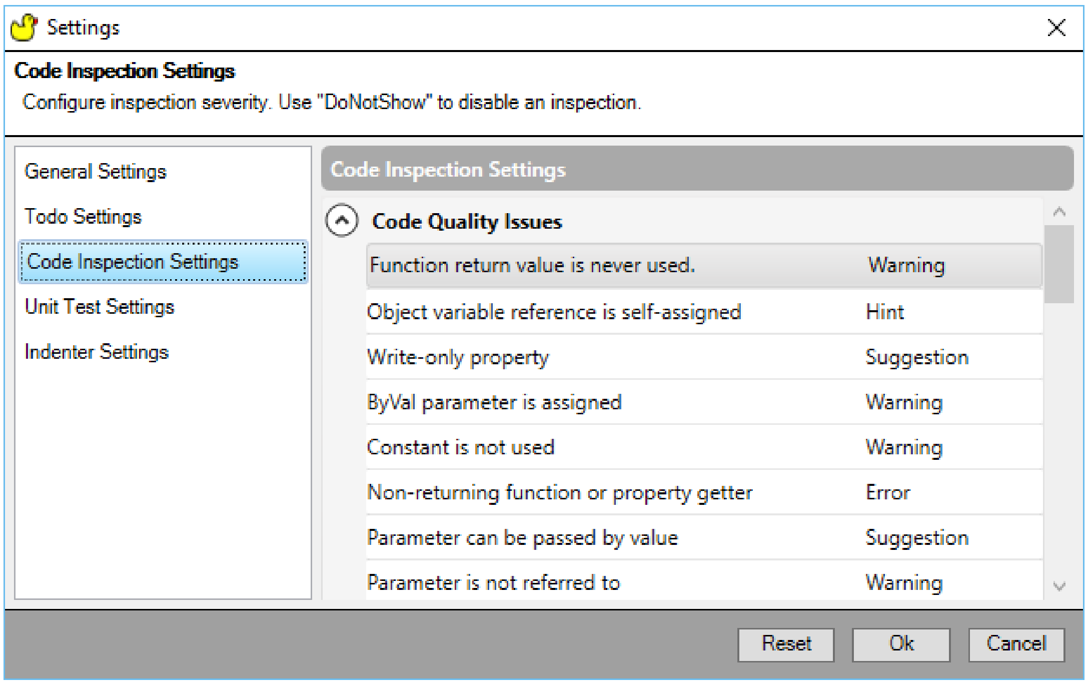
Return value for member 'GetFoo' is never assigned

This is likely a bug. The return value of a function or property getter must be assigned before exiting, otherwise the program will not be working with expected results. If a function has no meaningful return value, consider declaring it as a 'Sub' procedure instead.

VBAProject.Sheet1 L4C1 - L6C13

The bottom panel contains information about the selected inspection result, and fix-all links that always use the first quick-fix in the “Fix” menu. *Disable this inspection* turns the inspection’s “severity” to *DoNotShow*, which effectively disables the inspection.

You can access inspection settings from the **Rubberduck | Settings** menu in the main commandbar, or you can click the *settings* button in the inspection results toolwindow to bring up the settings dialog:


Settings

Code Inspection Settings

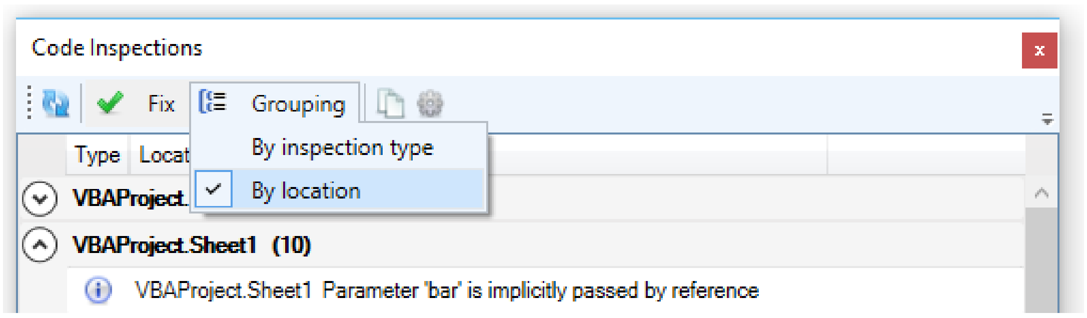
Configure inspection severity. Use "DoNotShow" to disable an inspection.

Function return value is never used.	Warning
Object variable reference is self-assigned	Hint
Write-only property	Suggestion
ByVal parameter is assigned	Warning
Constant is not used	Warning
Non-returning function or property getter	Error
Parameter can be passed by value	Suggestion
Parameter is not referred to	Warning

If you like using the **Call** keyword, you can easily switch off the inspection for it from there.

The **Copy** toolbar button sends inspection results into the clipboard so they can be pasted into a text file or an Excel worksheet.

As with similar dockable toolwindows in Rubberduck, the way the grid regroups inspection results can be controlled using the *Grouping* menu:



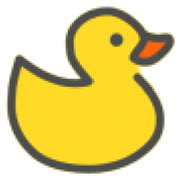
VBAProject.Sheet1 (10)

VBAProject.Sheet1 Parameter 'bar' is implicitly passed by reference

The *refresh* button causes a re-parse of any modified module; whenever parser state reaches “ready”, the inspections run and the grid refreshes – just as it would if you refreshed from the *Rubberduck* command bar, or from the *Code Explorer* toolwindow.

To be continued…

Posted in open-source, rubberduck, vbaTagged add-in, code analysis, parsing, pre-release, rubberduck, v2.0, vba, vba tools, vbe



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*

2 thoughts on “VBA Rubberducking (Part 2)”

1. **VBA Rubberducking (Part 3) – Rubberduck News** *May 18, 2016 Reply*
   […] Part 2 covered the code inspections. […]
2. **VBA Rubberducking (Part 4) – Rubberduck News** *May 28, 2016 Reply*
   […] Part 2 covered the code inspections. […]