OOP Battleship Part 1: The Patterns

Posted on August 28, 2018August 28, 2018 by Rubberduck VBA

About OOP

If you've been following this blog, you know that VBA is indeed very capable of "real" object-oriented code, regardless of what "real programmers" say about the language.

So far I've presented snippets illustrating patterns, and tiny example projects – the main reason I haven't posted recently is, I've been busy writing a VBA project that would illustrate everything, from factory methods to unit testing and *Model-View-Controller* architecture. In this blog series, you will discover not only that VBA code can be very elegant code, but also *why* you would want to take your skills up to the next level, and write object-oriented code.

You may have been writing VBA code for well over a decade already, and never felt the need or saw a reason to write your code in class modules. Indeed, you can write code that *works* – OOP will not change that. At one point or another you may find yourself thinking "well that's nice, but I'll never *need* to do any of this" – and you very well might be completely right. Think of OOP as another tool in your toolbox. OOP isn't for throw-away code or small, simple projects; OOP is for *large* projects that need to *scale* and be maintained over the years – projects you would show to a programmer in your IT department and they'd go "but *why* are you doing this in Excel/VBA?" …and of course the reason is "because that's the only tool you guys are letting me use!" – for these projects (and they exist, and they're *mission-critical* in every business that have them!), the *structure* and *architecture* of the code is more important than its *implementation details*; being *easy to extend* is more important than everything else: *these* projects are the projects that will benefit the most from OOP.

Object-Oriented VBA code is *much* easier to port to another language than procedural VBA code, *especially* with proper unit test coverage – which simply can't be done with traditional, procedural code. In fact, OOP VBA code reads very, very much like plain VB.NET, the only difference being the syntactic differences between the two languages. If your *mission-critical* VBA project ever falls in the hands of your IT department, they will be extremely grateful (not to mention *utterly surprised*) to see its components neatly identified, responsibilities clearly separated, and specifications beautifully documented in a thorough test suite.

Is OOP necessary to make a working Battleship game in VBA? Of course not. But taking this Battleship game as a fun metaphor for some business-critical complex application, OOP makes it much easier to make the game work with the human player on Grid1 just as well as on Grid2, or making it work with an AI player on *both* Grid1 and Grid2, or making different difficulty levels / strategies for the AI player to use, or trashing the entire Excel-based UI and making the game work in Word, Access, or PowerPoint, or all of the above… with minimal, inconsequential changes to the existing code.

Any of the above "changing requirements" could easily be a nightmare, even with the cleanest-written procedural code. As we explore this project, you'll see how adhering to the SOLID (https://en.wikipedia.org/wiki/SOLID) OOP principles makes extending the game so much easier.

But before we dive into the details, let's review the patterns at play.

PredeclaredId / default instance

I've covered this before, but here's a refresher. I find myself using this trick so often, that I've got a `StaticClass.cls` class module readily available to import in any project under my C:\Dev\VBA folder. The file looks like this:

```
VERSION 1.0 CLASS
BEGIN
MultiUse = -1 'True
END
Attribute VB_Name = "StaticClass1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
```

The `VB_PredeclaredId = True` attribute is the important part. With this attribute on, the class now has a *default instance*. What's critical is to avoid storing *instance state* in this *default instance* (see UserForm1.Show (https://rubberduckvba.wordpress.com/2017/10/25/userform1-show/)). But for pure functions such as *factory methods*, it's golden.

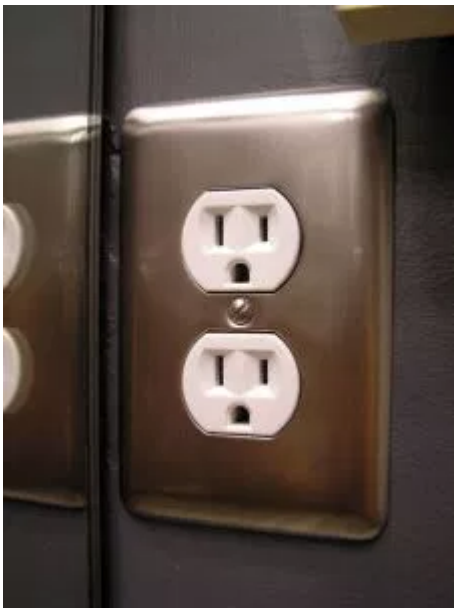
Under the hood, every single object is given an ID: when you `New` up a class, you create a new object ID. When a class has this attribute set to `True`, VBA automatically *pre-declares* an ID for an object that's named after the class itself.

Interfaces

Perhaps the single most powerful (yet underused) feature of VBA: the `Implements` keyword makes an instance of a class able to present different public interfaces to its clients. This allows us to have public *mutators* on a class, and yet only expose public *accessors* to client code that is written against an

interface. More on that below.

Think of an *interface* as a 110V power outlet.



It doesn't care what it's powering, so long as it fulfills the contract: any device that operates on a standard North American 110V power outlet can be plugged into it, and *it's just going to work*, regardless of whether it's a laptop, a desktop, a monitor, or a hairdryer.

An interface is a *contract*: it says "anything that implements this interface must have a method that does {thing}", without any restrictions on *how* that {thing} is actually implemented: you can swap implementations at any given time, and the program will happily work with that implementation, unaware and uncaring of the implementation details.

This is a very powerful tool, enabling *polymorphism* – one of the 4 pillars of OOP. But strictly speaking, *every single object* exposes an interface: its public members *are* its interface – what the outside world sees of them. When you make a class implement an interface, you allow that class to be accessed through that interface.

Say you want to model the concept of a *grid coordinate*. You'll want to have `X` and Y properties, …but will you want to expose `Public Property Let` members for these values? The `GridCoord` class can very well allow it, and then the `IGridCoord` interface can just as well deny it, making code written against `IGridCoord` only able to *read* the values: being able to make something read-only through an interface is a very desirable thing – it's the closest we can get to *immutable types* in VBA.

In VBA you make an interface by adding a class module that includes stubs for the public members you want to have on that interface. For example, this is the entire code for the `IPlayer` interface module:

```
'@Folder("Battleship.Model.Player")
Option Explicit

Public Enum PlayerType
HumanControlled
ComputerControlled
End Enum

'@Description("Gets the player's grid/state.")
Public Property Get PlayGrid() As PlayerGrid
End Property

'@Description("Identifies the player class implementation.")
Public Property Get PlayerType() As PlayerType
End Property

'@Description("Attempts to make a hit on the enemy grid.")
Public Function Play(ByVal enemyGrid As PlayerGrid) As IGridCoord
End Function

'@Description("Places specified ship on game grid.")
Public Sub PlaceShip(ByVal currentShip As IShip)
End Sub
```

Anything that says `Implements IPlayer` will be required (by the VBA compiler) to implement these members – be it a `HumanPlayer` or a `AIPlayer`.

Here's the a part of the actual implementation for the `AIPlayer`:

```
Private Sub IPlayer_PlaceShip(ByVal currentShip As IShip)
this.Strategy.PlaceShip this.PlayGrid, currentShip
End Sub

Private Function IPlayer_Play(ByVal enemyGrid As PlayerGrid) As IGridCoord
Set IPlayer_Play = this.Strategy.Play(enemyGrid)
End Function
```

The `HumanPlayer` class does something completely different (i.e. it does nothing / lets the view drive what the player does), but as far as the game is concerned, both are perfectly acceptable `IPlayer` implementations.

Factory Method

VBA doesn't let you parameterize the initialization of a class. You need to first create an instance, *then* initialize it. With a *factory method* on the *default instance* (see above) of a class, you can write a parameterized Create function that creates the object, initializes it, and returns the instance ready to use:

```
Dim position As IGridCoord
Set position = GridCoord.Create(4, 2)
```

Because the sole purpose of this function is to *create* an instance of a class, it's effectively a *factory method*: "factory" is a very useful OOP pattern. There are several ways to implement a factory, including making a *class* whose sole responsibility is to create instances of another object. When that

class implements an interface that creates an instance of a class that implements another interface, we're looking at an *abstract factory* – but we're not going to need that much abstraction here: in most cases a simple *factory method* is all we need, at least in this project.

```
Public Function Create(ByVal xPosition As Long, ByVal yPosition As Long) As
With New GridCoord
.X = xPosition
.Y = yPosition
Set Create = .Self
End With
End Function

Public Property Get Self() As IGridCoord
Set Self = Me
End Property
```

The `GridCoord` class exposes `Property Let` members for both the `X` and `Y` properties, but the `IGridCoord` interface only exposes `Property Get` accessors for them – if we consistently write the client code against the "abstract" interface (as opposed to coding against the "concrete" `GridCoord` class), then we effectively get a read-only object, which is nice because it makes the *intent* of the code quite explicit.

Model-View-Controller

This architectural pattern is extremely widespread and very well known and documented: the *model* is essentially our *game data*, the *game state* – the players, their respective grids, the ships on these grids, the contents of each grid cell. The *view* is the component that's responsible for *presenting* the model to the user, implementing *commands* it receives from the *controller*, and exposing *events* that the controller can handle. The *controller* is the central piece that coordinates everything: it's the component that tells the *view* that a new game should begin; it's also the component that knows what to do when the *view* says "hey just so you know, the user just interacted with cell F7".

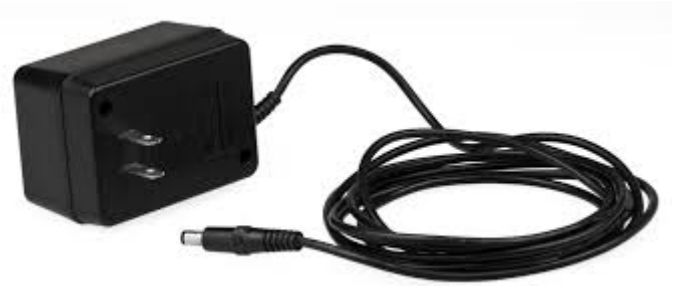
So the controller knows about the model and the view, the view knows about the model, and the model knows nothing about no view or controller: it's just data.

Adapter

The *adapter pattern* is, in this case, implemented as a layer of abstraction between the *controller* and the *view*, that allows the former to interact with anything that implements the interfaces that are required of the latter. In other words, the *controller* is blissfully unaware whether the *view* is an `Excel.Worksheet`, a `MSForms.Userform`, a `PowerPoint.Slide`, or whatever: as long as it respects the *contract* expected by the controller, it can be the "view".

Different view implementations will have their own public interface, which may or may not be compatible with what the controller needs to work with: quite possibly, an electronic device you plug into a 110V outlet, would be fried if it took the 110V directly. So we use an *adapter* to conform to the

expected interface:



Or you may have taken your laptop to Europe, and need to plug it into some funny-looking 220V outlet: an *adapter* is needed to take one interface and make it compatible with another. This is quite literally *exactly* what the adapter pattern does: as long as it implements the `IViewCommands` interface, we can make the *controller* talk to it.

Posted in *OOP*, *rubberduck*, *tutorials*, *Uncategorized*, *unit-testing* *Tagged* *design-patterns*, *oop*, *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*

4 thoughts on “OOP Battleship Part 1: The Patterns”

1. **SmileyFtW** *August 28, 2018* *Reply*

   Great stuff, Mathieu, and thanks for doing this. One thing that I would greatly appreciate is being able to have the full code module(s) text available for each topic as you post it. I struggle to see the entire picture and I think having the full code along with the actual names of the classes, etc. would help greatly. Posting as either plain text downloads or as the actual exported modules would work.
   Again, many thanks!
   David

   Rubberduck VBA *August 28, 2018* *Reply*

   For this introductory post I’m only glossing over the concepts; I’m planning to make the whole workbook / VBA project available for download in every other part of this series though (and I’ll probably update this one with a download link too)… I’m still implementing the game at this point though, so I’d rather publish the finished, complete code =)

   Rubberduck VBA *September 4, 2018* *Reply*

   I published parts 2 & 3; here’s the download link for the game: https://www.dropbox.com/s/00pefak6k47vidr/Battleship%20%28WorksheetView%29.xlsm?dl=0 — enjoy!

2. **VBA+OOP: What, When, Why – Rubberduck News** *August 29, 2018 Reply*
   […] I’m writing a series of articles about a full-blown OOP Battleship game, and generally speaking keep babbling about OOP in VBA all the time, it occurred to me that I might […]