# Title page

Are there any software engineers here? (I might get away with this!)

Has anyone here had to work on some code that is really awful?

Has anyone here had to work on some code that is awful, but you suspect it started out OK when it was first written?

It’s more-or-less OK at the outset, isn’t it? But then it dies from a thousand cuts.

# Evolution graphic

PAUSE

The profession of SE is primarily concerned with change.

# Assembly code

History lesson!

Machine code – might be programmed with mechanical switches, punch cards, etc. Assembly language is the use of human-readable keywords to write machine code, which is then run through an assembler to assemble source files and to convert into binary machine code.

We call this the “imperative” programming paradigm – you execute a sequence of statements.

# C

C is a “Procedural” language – you can define sections of code that you can invoke by name, and pass in parameters. This is a big win for readability.

Complex data types are accessed through pointers. A pointer is just a memory address. You manually allocate a chunk of memory and get back a pointer to the start of the chunk. It’s up to you what you do with it. If you want to store a string, you allocate enough memory for as many characters as you want. To get the characters in the string, you start with the pointer – which is just a number – then keep adding 8 to dereference each character. If you go past the end of the string, who knows what you’ll get?

Once you’re done with your string, you need to manually deallocate the range, or you have a memory leak.

# Struct graphic

C does not have classes and objects, but it does have structs.

A struct is a data-only object that maps fields of fixed size to offsets from a pointer. It’s a bit like an Access database record.

The start of this object is a known memory address, and the size of each field is known, so to get a field – to dereference it - you just perform arithmetic.

If our pointer is 0x0AC73800, then our first field is at 800, our second field is at 804, our third field is 808, and so on.

# Heap graphic

Nowadays, we use languages that provide memory management. You don’t care about memory addresses.

Instead of pointers, you have references. The runtime sorts out where the actual pointers are. I’m showing you this graphic, but you literally don’t care, because you can store something and get it back without thinking any more about it.

# Java / C#

Does anyone here regularly think about deallocating memory? If so, how do you accomplish it? (Or – No, you don’t have to) Yep – drop the reference and walk away. Java and C# have garbage collectors.

This is achieved because these languages are not compiled to machine code – they are compiled into an intermediate code which runs in a runtime engine. The runtime engine provides memory management. A compiler COULD insert subroutines for this and compile to machine code – but that’s not how Sun and Microsoft designed the languages.

Memory management tag-teams with scopes. In C, a memory allocation exists until it is manually deallocated. In C#, a variable exists until it falls out of scope. Typical scopes are instance-scoped – an object’s data dies with the object – and method-scoped – a variable ceases to exist when the enclosing method returns. This makes it much easier to avoid memory leaks.

In Java and C#, references are strongly-typed. If you declare a variable as a string, no bug can make it not-a-string. The code won’t compile if you try to make it something else. (Sometimes you can get this to compile, but it will throw at runtime.)

Java and C# are object-oriented. Instead of executing sequence of statements, including passing data into procedures, you primarily define objects and make them interact with each other. An object is one or both of state (data) and behaviour.

# Access modifiers graphic

Members of a class can be declared public or private. This makes them visible or invisible outside the object. What you can see of and do with the object is restricted.

As long as he does the finance stuff, it’s no business of anyone else what Businessman chooses to spend his money on.

Even if it’s not a traditionally masculine toy.

And it’s only £19.99 at Toys’R’Us.

And he still has his Christmas money to spend.

And Aunt Jean told him he could SPEND IT ON WHATEVER HE WANTED, DAMMIT!

But by making our choice of toy private, we give ourselves the flexibility to change it later, if we decide we want to move from My Little Pony to Furbies.

We also make it easier for other devs to use our code, by removing options that aren’t helpful. It would be frustrating for a developer to spend half an hour flicking through an underwear catalogue only to find out that setting the underwear to something with frills does not have the desired effect on the Finance Stuff.

We say that the *interface* of BusinessMan defines AttendMeeting(), DoFinanceStuff() and BusinessSuit. And now we can look at our first principle…

# Code to an interface, not an implementation

Anything you expose from any entity forms an interface. An interface forms a contract with the world – if you only access these members, I will keep the behaviour consistent.

All the rest of it is internal mechanism – and that means that you can change it any time you want. And this doesn’t just mean changing how the object works under the hood – it might mean swapping the object out for another one that implements the same interface.

If you take the covers off and access the mechanism directly, it serves you right if you get your fingers jammed. Some of that underwear is really tight.

The flip side of this is that, when we write code that others will interface with, we should think carefully about what our interface should expose, because that’s what we *don’t* want to go changing later.

It’s all about the interfaces. Get the interface right, and you have your design. Get it wrong, and you’ll be fighting it further down the line.

In Java and C#, interfaces are first-class citizens and have their own keyword. In Powershell, this principle mainly applies to:

* what you export from a module
* what you accept in a parameter block
* what you return from a function
* to a lesser extent, what members of a class you mark as “hidden”

All of this is dynamic. You can’t enforce a return type from a function.

In Powershell classes, you also cannot - strictly speaking - define an interface or a private member. But I’ll show you how to fudge that later.

# DSC / YAML / SQL

Honourable mention – the “declarative” paradigm. You define the input state, and set the machine going. You do not write statements to be executed in sequence.

# LISP / R / F#

Functional programming is a subset of declarative programming in which, unlike YAML, you also declare the functions that work on the data.

It’s still not about executing statements in order – it’s about mapping input to output.

But I won’t go into this because it’s extremely tangential to powershell!

# Procedural vs OOP

In software, you have to model the problems that you want to solve.

OOP has become the dominant paradigm for because modelling components as objects with state and behaviour lets you think at a higher level about the problems you need to solve. The code becomes more readable.

I’d better explain what “polymorphism” means: it is the idea that you can swap objects to change behaviour. For example, .NET provides a number of StreamReaders in the base class library (BCL). You have MemoryStreamReaders, FileStreamReaders, TcpStreamReaders, XmlStreamReaders. All of these provide a unified way to get the data.

If we swap a FileStreamReader out for a network protocol streamreader, we are changing behaviour using polymorphism.

In Java and C#, a requirement for this to work is that all the possible objects implement the same interface. So, if you hold something with that interface type, the actual object could take many forms. Languages like Python and Powershell aren’t this strict.

I’ll be coming back to this later.

# …Powershell

About the weak typing – you absolutely can blow up your code by trying to do string things to a WMI object, for example. There is no compiler stage in the development process to catch type errors. The best you get is IntelliSense suggestions.

You can declare a variable as a particular type – e.g. in param blocks – but this is runtime checking and all it means is you get a helpful exception immediately instead of a weird one later.

This may be a good time to warn you all that “Powershell” and “Software architecture” are not commonly mentioned in the same breath, and that’s because the dynamic nature of them give you less help as a software architect.

In fact, I’m giving this talk because there’s been very little written about software engineering in Powershell. And some of what I’m going to show you is a bit of a hack, because the language doesn’t naturally support a strongly-defined architecture. I’ve come to believe that Powershell is not inherently a good choice for complex projects. The reason we’re sitting here is because Powershell works for us and we know it, and one of the reasons for choosing a particular language is whether your team has skills in the language. Therefore, applications are going to get built in Powershell, and that’s why I want to get my thoughts out on how to engineer software in it.

# Software Engineering

Software Engineering is a profession that is primarily concerned with change.

* New acquisitions
* Regulatory change, business process change
* New apps and providers

You can write something that works, now – but to do a good job, you may have to write something that is also evolvable.

“Evolvability” implies more direction than just “change” – it suggests that you have an architecture at the beginning and, two or three years on, you still have an architecture. It may not be the same architecture, but there is still coherence to the product.

If we are writing projects of some size and we do not concern ourselves with the practice of SE, we are in danger of ending up, either initially or after a few years’ worth of change, with spaghetti code, where further change becomes expensive and demoralising.

# Design patterns

The tools we use to perform software engineering are:

* Our development environments
* Our skills in the language we use
* Our project management toolset (Agile, Jira)
* A set of guiding principles
* Design patterns

Design patterns are conceptual solutions. The term was first applied to software in 1994, when the Gang Of Four described 29 of them. Now you could probably find over 100. They are language agnostic, although some naturally fit better in certain paradigms than others. (In fact, they usually assume OOP.) And if you are trying to explain your code and you name the patterns you use, you can convey a lot of information quite concisely.

# Code\0. Problem statement

## SampleModuleManifest.psd1

This is the focus of the demo – you guys have all seen this, I take it? This is a module psd1, this field (RequiredModules) takes a mixed array of strings and hashtables, the hashtables have to have ModuleName and a property representing the version.

If you put a dependency in RequiredModules, Powershell will attempt to autoload it if it’s not present, and if that fails, the dependency is not met and the rest of your module won’t load.

That means that, if you can’t rely on Autoload, the order is important when you import modules.

## Download-ModuleAndGetDependencyOrder.ps1

This is an approximation of a solution we use at my employer. We distribute our code through releases on Github. So in addition to more typical githubby things, we also have functions relating to downloading modules from repos. This sample is used in our flagship app, which is a platform to run scripts from github on customer devices.

This function runs in the service layer. It does two things: it downloads all the required modules from github, and it returns them in reverse order. So it gives you an import order for the modules, dependency modules first, up to the module you start with.

Now our problem is this: our licensing team is getting much more aggressive with github licenses, and we can’t distribute our code through github any more. We need to move to a nuget feed and incorporate that with our Jenkins pipeline. So we look at our function and go – oops – almost every line touches Github. And worse, our param block is completely geared towards github too, so we’re going to have to change everything.

And this isn’t the only problem, because this is only one place where we touch github and module dependencies. We also use it in our build process in code that’s structured similarly but only outputs the dependencies in a pretty format for visual inspection, and of course we’ve got our homebrew package manager. All of these are implementing broadly the same approach but with no shared code. So here’s our next principle:

# Don’t Repeat Yourself (DRY)

So, we are going to reimplement this code in a generalised way, so that it can be used for both Github and nuget, and so that we can use the same code everywhere. And to do that, we’re going to need to separate the Github code from the module dependency code.

# Separation Of Concerns

Separation Of Concerns says that an object – and I use the word “object” in a loose sense – should do only one thing. Our current solution, as we’ve just seen, does two things – it interacts with Github, and it walks through module dependencies.

The dependency code is simple function recursion, and that vanishes when the function completes, leaving us with nothing, so I think it’s appropriate to build a data structure for it so we have something we can work with in future.

You may have one, none or many modules that you require in a module, and each required module may in turn have one, none or many modules that it requires, so we end up with a tree structure, with the module we first touch at the root, and all the required modules as nodes along the way. A module is a leaf if it does not require any modules.

How do we build a tree?

# Code\1. Simple implementation\

## Node.ps1

We build the nodes, don’t we, and then we make sure that each node has a parent property and a property that’s a collection of children, and then we plug the nodes together. Then our module dependency function will take the input to create the first node, then recursively do the children, then output a tree object, and we’ll use that to get our dependency order and everything else that we do everywhere else.

But these nodes don’t tell us anything about modules, so we’re going to add a module name property, and modules have a guid so we’ll add that, and modules have a version so we need to add that – just kidding, we’re going to use something that exists already. Does anyone know what class we should use to represent a module?

## ModuleSpecification.cs

We’re going to use a class called ModuleSpecification, which already has those properties.

And, it gets better. When you import a module with a psd1, anything in RequiredModules gets converted by the powershell engine into one of these ModuleSpecification objects. And if we look at the constructors, we can see that they align with the possible things we can put into the RequiredModules section of a .psd1 file.

## ModuleSpecification.psm1

We get the properties for free because we inherit them, we still add our Parent and Children and we still add our PrintTree method.

We add constructors that match the existing ones, but all we do is pass the input parameters on to the constructor we’ve inherited.

## Get-ModuleDependency.ps1

We give ourselves a little function that manually constructs the tree. And if we run it, hey presto, Roof depends on Walls, Walls depends on Foundation.

We’ve built this as a separate function for two reasons:

* There’s some other stuff to cover before we could think about doing anything clever
* We may want to make our code available as a function, if that’s more convenient

OK, that’s a bare minimum solution for the tree…

# Code\2. Separate ModuleFetcher class\

ModuleSpec is unchanged.

## ModuleFetcher.psm1

We’ve created a class that mocks out downloading, importing and returning a module.

We create a dummy PSModuleInfo. We give it the name that the requester asked for, we fudge it with reflection to give it something in the RequiredModules field, and we output the module we created.

## Get-ModuleDependency.ps1

Now we’re going to take a parameter of our ModuleSpec class. That same object is what’s returned at the end.

In our test code, we’re going to pass a string to that parameter, so we’re going to be leveraging Powershell’s helpfulness with types. But we know we’ve already got a constructor that takes a string name.

We create the fetcher, we call its method and get back a PSModuleInfo. We iterate over each RequiredModule and call our function recursively.

We’ve now separated out code to fetch modules form code to represent module dependencies. Our function knows about ModuleFetcher and ModuleSpec, but ModuleSpec is unchanged from before we wrote ModuleFetcher, and in ModuleFetcher there is no mention of ModuleSpec at all.

# Classes (or modules) should be loosely coupled

Every property or method or module function that you touch from another class or module increases the coupling between those two entities. That makes it harder to change in the future.

So let’s get past this crappy mock. I have created a module structure on disk…

# Code\MockModuleRoot\

Each of these contains a versioned module manifest.

## Human\1.0\Human.psd1

Human requires Cow, Pig and Cabbage, all at version 1.0.

# Code\3. FileSystemModuleFetcher class\

## MockModuleFetcher.psm1

We’ve taken ModuleFetcher and renamed it MockModuleFetcher. We’ve made it inherit from ModuleFetcher, so what used to be ModuleFetcher is now an abstract base class. “Abstract” means you can’t make an instance of it. It exists only to provide an interface.

## ModuleFetcher.psm1

We’ve got this abstract base class now. This means that anything that inherits from ModuleFetcher is going to have a GetModule method which takes a ModuleSpecification and returns a PSModuleInfo.

Remember how I said that Python and Powershell are a bit too flexible to really support certain OOP concepts? This is an example. If this were C# and I’d defined this class as an interface, not only would the compiler guarantee that I had written a proper implementation in every subclass, but Visual Studio would pre-populate it for me. As it is, I can skip it and the code will run. However, when you fail to implement this method in your subclass, it will give you this NotImplementedException, which should be picked up in testing and will tell you exactly what you need to do. So we’ll say that the value of the abstract base class is more in documentation than in enforcement. And we’ll remember that design patterns are primarily a conceptual thing.

This means we can also create:

## FilesystemModuleFetcher.psm1

This adds a ModulePath property, that’s expected to be a folder on disk. In this class, when we call GetModule, the method searches the root folder we defined and returns a module from disk.

These two classes respect the…

# Liskov Substitution Principle

Because the base class implements a method with a particular name, parameter type and return type, and the derived class also implements that method signature, we can declare our variable as the abstract base class but actually use either the mock or the filesystem class.

## Get-ModuleDependency.ps1

We still declare the $Fetcher variable as ModuleFetcher, but we’ve added a switch parameter to decide if this is going to be MockModuleFetcher or FilesystemModuleFetcher.

The behaviour we’re going to get does not come from a bunch of if/else statements in the function body. Yes, there is one where we decide which class to create, but run with me on this one, it will become clear. As you can see, once we’ve instantiated the object of our choice, our method call to GetModule is the same.

* The contract for that method comes from the base class.
* We declare our variable as the base class. Remember, we can’t create an object of the base type because it’s abstract.
* But we can create either type of derived class and put it in our variable.
* Then we can call the method that the contract provides, and which behaviour we get is a result of which type of object we created. That’s polymorphism.

In this case, the Get-ModuleDependency function is just a framework for slotting in an instance of ModuleFetcher. That piece of the function’s behaviour doesn’t come from the function itself, it’s what we call “composed” from the object that provides the behaviour. And this is an example of

# Strategy pattern

This is an object that has pluggable behaviour.

The behaviour that you can plug in is constrained – it’s not a free-for-all. But you can see that we’re only a short step away from adding Github behaviour or Nuget behaviour at this point. So we have gained a lot of freedom to change the Get-ModuleDependency function.

We assessed what might change, and structured the code to make that change easy.

But there’s still a slight problem!

* Change switch if/else to if/elseif/else {throw NotImplemented}

To add a github or a Nuget fetcher, we have to edit the function to create the new object, we have to update the param block to select that code branch, and that means we have to edit the code that calls this function.

This object still controls what strategies we can use. The control is not with the client.

To be fair, it is not difficult to change a Powershell function, but it would be nice to be able to swap in behaviour without making any changes at all. So we are going to use…

# Dependency Injection (DI) pattern

We aren’t going to create whatever module fetcher we want in the function any more. It doesn’t scale, and we don’t want to let one function dictate how we write the rest of our code, so we’re going to pass our choice of ModuleFetcher into the param block instead.

DI can be used for other things that Strategy. For example, if you have a Logger class, you aren’t necessarily changing behaviour or using polymorphism if you only have one Logger, but it’s still a good thing to pass in with DI.

But I’m not going to bang on about that any further, let’s take a look.

# Code\4. Dependency injection\

## Get-ModuleDependency.ps1

We’ve got rid of the switch, which could only do one thing, and replaced it with a param of the base class type.

When we call this function, we create the fetcher we want and pass it in.

And we can still pass either kind of fetcher.

Now let’s make the tree do something for us and import our module.

## ModuleSpec.psm1

We’ve added a GetImportOrder method that works a lot like the PrintTree method – we have a public version that takes no parameters, creates a variable, and passes it to the hidden version of itself, which does the work recursively.

## Get-ModuleDependency.ps1

So does it work? We get the list in the correct order, but import fails, because we haven’t implemented any version code.

# Code\5. Implement IsMetBy\

## ModuleSpec.psm1

We’ve implemented the constructor that takes a PSModuleInfo.

We’ve created an IsMetBy method, that checks that the module we’re testing has the right name, guid and version, depending on whatever is in the ModuleSpec.

## FilesystemModuleFetcher.psm1

We call the IsMetBy function while we’re searching for modules on disk. When we do this, we’re introducing a dependency, which is a shame, but not too bad. You can’t have code with no dependencies at all. This is still under control.

My team uses exact version pinning. If you state that you want version 1.6.1 in a RequiredModules field of a psd1, we don’t accept 1.6.2. And that’s very unlikely to change. But if it did, and we wanted to swap freely, how could we do it?

We could create a strategy object that implements IsMetBy, and we could inject it into the module fetcher. And if we ever needed to we could inject it into ModuleSpec as well. And that would reduce the coupling and give us that flexibility back. But seriously, are we ever going to move away form an exact version pinning strategy? I doubt it, so

# YAGNI

Don’t get carried away early on with generalising things that aren’t going to change. That problem belongs to a tomorrow that may never come.

# Refactoring

But if, later on, you discover that you do need to change it, that’s the time to refactor.

A refactor can be big, like we’re doing in this demo, or small, where you move chunks of code into separate methods.

You are refactoring when you add a function to a module and you realise that a lot of the code duplicates something in an existing function, so you pull that duplicated code out into a new private function.

I am not going to implement a github function or a nuget function for you all. I believe I’ve shown the way, and most of you know how to do the rest. But there is one more thing I’m going to do in this demo. I don’t like that the dependency tree object, our ModuleSpec, is also responsible for formatting. We haven’t done a good job on formatting and it’s inflexible. Besides, we only want the tree to be a tree, so we’re going to follow Separation Of Concerns and move the formatting into a ps1xml.

# Code\6. Separate formatting\

## ModuleSpec.ps1xml

A very simple format.ps1xml that shows name, version and guid. But the table entry for Name is a scriptblock that applies indentation proportional to how deep in the tree the row entry is. (You guys know about multiplying strings by ints, right? It repeats the string int number of times?)

## ModuleSpec.psm1

We’ve got a hidden \_Generation field. An uninitialized int is 0 and that’s a valid value, so we initialise it with -1. Then if we’ve got -1, we know we need to work it out, so we recurse upwards through the parent property and add 1 each time.

We delete the PrintTree garbage.

Instead we create a ToList method which works pretty much the same.

We reimplement GetImportOrder to simply reverse the list.

## Get-ModuleDependency.ps1

And now when we call ToList we get a nicely formatted table.

That’s the end of the code walkthrough.

# Recap

* We thought about what kind of change we might have to support
* We came up with an architecture that supports that change
* We separated the concerns
* We implemented one of the variants as a Strategy object
* We injected our strategy object into our main object (by which I mean the function)
* We equipped our tree structure with methods to both format and to get an import order
* We have a clear path forward

# Appendix – other design patterns

# 1. Adapter

The adapter is when you have an object of one type and you need it to be of another type. An adapter translates one interface into another.

You have all your managed devices in Tivoli. It’s got your IPs, credentials, and SKUs. You have a whole bunch of modules that expect these properties in Tivoli’s format. But ServiceNow licensing is cheaper! So now you’ve got to dedicate a resource to the migration, and your backlog is getting worse, and the last thing you need is to rework all your code because IP addresses are now ip\_address instead of ipaddress.

So you’d probably write a ConvertTo-Tivoli that takes your ServiceNow objects and makes them Tivoli-compatible. That’s an Adapter.

You might do this just by adding alias properties. Alternatively, you might create a PSObject where one of the properties is the original object, and the rest are scriptproperties that reference the original object. There’s a lot of ways to do it.

The best bit is that it solves coexistence without you having to accept both types of object everywhere. Can you imagine how many conditionals you would have to add?

## Code\Appendix 1. Adapter pattern\Param alias.ps1

I’m not sure if this counts as an example of the Adapter pattern, but I’d argue it does. GCI gives you a FileSystemInfo object. If we add an alias to our function parameter, then it adapts the FileSystemInfo object to the expected parameter input.

Please remember that the vast majority of the literature about design patterns is written for OOP, not procedural languages.

# 2. Façade

Façade is very similar to Adapter; the difference is in purpose. An Adapter is where you wrap an object to make it conform to the interface expected somewhere else; a façade is where you wrap an object to make it simpler to use.

It’s arguable that any API client module is a remote façade, where you have a convenient local interface to a remote process.

But the real reason I bring this up is not because you might write one, but because Powershell does it an enormous amount.

The objects you get back from Get-Service are facades for the output from the service controller API. Likewise, Get-Process gives you facades for unmanaged objects from the Win32 API. Almost every network related cmdlet built in is a façade for WMI. And that’s Powershell’s strength for systems administration; it gives you a lot of facades for objects from otherwise inconvenient APIs.

# 3. Proxy

The proxy pattern is where you wrap one object with another that mimics it, but provides some extra functionality.

If you alter the functionality, you’re arguably implementing the Decorator pattern, not the Proxy pattern. But these two patterns are anyway not as distinct in procedural programming as they are in OOP.

## Code\Appendix 3. Proxy\Proxy for Import-Module.ps1

In this case, we faithfully implement all the parameters of Import-Module, and in the function body, we call the original Import-Module. But we add our own parameter too, and do some extra work based on that parameter.

I’ve used this quite a lot. Powershell makes it very easy.

I’ve used it without adding any extra params, just in order to set a default parameter value that isn’t present in the original command.

Our flagship app runs scripts on a remote device via a web API. The CLI is based on proxies. The proxies implement all the original parameters, plus an extra one to tell it what device to connect to. The function body just invokes the API and then formats the return from the remote device, so, apart from the delay, it’s just like running locally. That’s called a Remote Proxy.

# 4. Iterator

This isn’t one you’re likely to write, but you might end up using one day. Iterator is a pattern for stepping through a collection, that can work the same way for a large number of collections.

Quick tangent – why is it that you can index into a string, but if you pipe a string to ForEach-Object, it doesn’t iterate over the characters in the string?

Because string doesn’t implement IEnumerable. Everything else like List, Hashtable, Array, all these things do, and that’s how you can pipe them in Powershell.

IEnumerable defines one method: GetEnumerator(). Enumerator is what Iterator is called in .NET, and the powershell engine uses this every time you pipe a collection.

## IEnumerable.ps1

This is how it works to use an Enumerator.

So, we could then go and update our tree to:

* make it implement IEnumerable
* Implement a GetEnumerator() method
* Write a custom Enumerator class that knows how to step through our tree

Then we could pipe our module dependencies instead of calling ToList().

# 5. Decorator

Python handles decorators really well, there are libraries for it. Powershell doesn’t help you so much. However, it does give you transparent proxying out of the box.

*Highlight in IDE from Appendix 3*

$CommandInfo = Get-Command Import-Module

[System.Management.Automation.ProxyCommand]::Create($CommandInfo)

If we run this, we get a text function definition that transparently proxies another command. And the begin, process and end blocks are fixed points of reference. So we could use regex to snip at the begin block and insert some logging code. That would be a Decorator. And you could do it automatically on module load, if you have a global variable set.

OK, I really can’t cram any more in, so there’s one last point I’d like to leave you with:

# They’re more like guidelines…

Don’t be too rigid about adhering to principles or using patterns.

Sometimes you just have to break a rule to make the code work.