## 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

Machine code - mechanical switches, punch cards

Assembly language: human-readable machine code

"imperative" programming paradigm

## C

"Procedural"

Complex data types => pointers.

Pointer => memory address. Manually allocate and deallocate.

## Struct graphic

## Heap graphic

Runtime handles pointers, gives you references instead

Deallocated => garbage collected

Scoped

Loads of low-level work removed – can think more about the real problem

## Java / C#

References are strongly-typed.

Java and C# are object-oriented => 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 => 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 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.

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.

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"

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.

## LISP / R / F#

Subset of declarative programming => also declare the functions that work on the data.

It's about mapping input to output.

## Procedural vs OOP

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

Explain polymorphism

## …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

Evolvable

If we do not concern ourselves with the practice of SE => danger of spaghetti code, further change becomes expensive and demoralising.

## Design patterns

The tools we use to perform software engineering are:

- Our development and test 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.

# Code\0. Problem statement

## SampleModuleManifest.psd1

Show RequiredModules

Import order matters

## Download-ModuleAndGetDependencyOrder.ps1

This function runs in the service layer. Downloads all the required modules from github, and it returns them in reverse order.

Github licenses => move to NuGet

- Every line touches Github
- param block is geared towards github => interface change => change calling code

Also used in build process => structured similarly but only pretty format

Homebrew package manager.

So here's our next principle:

# Don't Repeat Yourself (DRY)

Generalise

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 => vanishes when function completes

Tree => How do we build a tree?

# Code\1. Simple implementation\

## Node.ps1

Parent and Children properties

This doesn't tell us anything about modules => add name, guid, version – NO!

Inherit. What to inherit from?

## ModuleSpecification.cs

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

Used by powershell engine when parsing a .psd1 file => constructors align to RequiredModules field

## ModuleSpecification.psm1

We get the properties for free because we inherit them

Chain constructors (don't need to do anything more)

### Get-ModuleDependency.ps1

Bare minimum solution for the tree…

## Code\2. Separate ModuleFetcher class\

ModuleSpec is unchanged.

### ModuleFetcher.psm1

A class that mocks out downloading, importing and returning a module.

### 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, but will have been fleshed out.

Pass in a string => constructor chaining

We've now separated out code to fetch modules from code to represent module dependencies.

- Function knows about ModuleFetcher and ModuleSpec
- ModuleSpec is unchanged
- ModuleFetcher has no mention of ModuleSpec

## 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

Renamed ModuleFetcher => MockModuleFetcher, made it inherit.

### ModuleFetcher.psm1

Now it's a base class. How can we instantiate? CAN'T. It's abstract.

Anything that inherits from ModuleFetcher => GetModule method

(In C#, Visual Studio would help us.)

This means we can also create:

### FilesystemModuleFetcher.psm1

Adds ModulePath property

These two classes respect the…

## Liskov Substitution Principle

Because the base class declares 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 base class but actually use either the mock or the filesystem class.

### Get-ModuleDependency.ps1

Behaviour does not come from if/else statements. There is one, but bear with me.

Once we've instantiated, our method call to GetModule is the same.

- Method contract comes from the base class.
- We declare our variable as the base class.
- 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.

Get-ModuleDependency function => 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.

Could add Github / Nuget behaviour.

Gained a lot of freedom. But only in a sensible direction.

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
- Add ModuleRoot param with parameterset

Add Github / Nuget fetcher => edit function => update param block => update calling code

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

So we are going to use…

## Dependency Injection (DI) pattern

Move dependency creation out of function.

It doesn't scale, don't want to let one function dictate how we write the rest of our code, so pass fetcher into param block instead.

DI can be used for other things that Strategy. For example, a Logger class (not behaviour change, not polymorphism, as only one class), but it's still a good thing to pass in with DI.

# Code\4. Dependency injection\

## Get-ModuleDependency.ps1

We've got rid of the switch => replaced it with a param of the base class type.

DI thru param block. How else? Module scope variable.

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

GetImportOrder method

## Get-ModuleDependency.ps1

List wo0rks, but import fails

No version code.

# Code\5. Implement IsMetBy\

## ModuleSpec.psm1

We've implemented the constructor that takes a PSModuleInfo.

IsMetBy method

## FilesystemModuleFetcher.psm1

In .ForEach, change [$true} to IsMetBy

Introducing a dependency, increasing coupling. Nvm.

We use exact version pinning. If we decide to change, how?

Strategy & inject it into fetcher. But seriously, will we ever? 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.

Not going to implement Github / Nuget in this demo.

One more thing. Tree object is also responsible for formatting. We haven't done a good job on formatting and it's inflexible.

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.

That's a refactor.

### 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.

## 6. Compound

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.