# 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

If you’re a developer, you can write something that works – but an engineer has 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.