The Monad Manifesto

• Annotated •

by Jeffrey Snover
as annotated by the
PowerShell Community

Snover art courtesy RunAsRadio.com

Table of Contents

The Monad Manifesto is the original Jeffrey Snover-authored document that results in the Windows PowerShell we know today. This version includes community-contributed annotations, notes, and expansions.

The Monad Manifesto - Annotated

by Jeffrey Snover as annotated by the PowerShell Community

This project is intended to preserve and annotate "The Monad Manifesto," a paper written by Windows PowerShell inventorJeffrey Snover at Microsoft in 2002. The idea for this project came from Pluralsight author Tim Warner, with the initial annotations being made by Tim and Microsoft MVP Don Jones.

The original Manifesto was a forward-looking document, predating the public release of PowerShell by around 4 years. In the years since PowerShell's 2006 release, the product has evolved substantially - but always around the broad brush strokes outlined in the Manifesto.

We felt that it was not only important to preserve the document for historical purposes, but also to annotate and expand upon the various concepts it introduces. We'll attempt to link to references for the now-real technologies that the Manifest predicted, and to provide contextual explanations around some of the Manifesto's directives.

You'll notice [1] footnotes in the text. These are a MultiMarkdown feature that aren't supported by our publishing platform, but they're meant to link to corresponding footnotes at the bottom of the page. In some cases, these are Jeffrey's original footnotes, and we've marked those with "ORIGINAL" to set them apart from footnotes we've added ourselves.

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or

MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective

- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

Chapter 1 - What is Monad?

Monad[1-1][1-2] is the next generation platform for administrative automation. Monad solves traditional management problems by leveraging the .Net Platform. From our prototype (though limited), we can project significant benefits to developers, testers, power users, and administrators. Monad leverages[1-6] the .NET Common Runtime to provide a powerful, consistent, intuitive, extensible and useful set of tools that drive down costs of administration and make the life of non-programmers a lot easier.

Monad consists of:

1. Monad Automation Model (MAM): An automation model based upon .Net classes, methods and attributes to produce Cmdlets.aspx).[1-3]
2. Monad Shell (MSH): A .Net based script execution environment for exposing Cmdlets as APIs command line tools and interactive programmable command line shell.
3. Monad Management Models (MMM): The set managed code base classes (or interfaces) to implement specific management scenarios and in-the-box administrative tools to execute those scenarios.
4. Monad Remote Scripting (MRS): A set of Web Service based components that allow scripts to be remotely executed on many machines[1-4].
5. Monad Management Console (MMC): A .Net based model and set of services for building management GUIs on top of MSH and exposing all GUI interactions as user-visible scripts[1-5].

This white paper presents the traditional approach to administrative automation, its strengths and shortcomings. Monad's new approaches are then articulated. An overview of the major components of Monad is then presented. A set of value propositions is then articulated for Monad's target audiences.

Notes:

1-1. (ORIGINAL) This is not a Windows PowerShell whitepaper nor is it an accurate description of how V1.0 works. This is a version of the original Monad Manifesto which articulated the long term vision and started the development effort which became PowerShell. Many of the elements described in this document have been delivered and those that have not provide a good roadmap for the future. The document has been updated for publication. Confidential information has been culled and examples are updated to reflect the current syntax. ↩

1-2. (ORIGINAL) Monads are Leibniz's term for the fundamental unit of existence that aggregates into compounds to implement a purpose. In this philosophy, everything is a composition of Monads. This captures what we want to achieve with composable management. More information on Monadology can be found at: http://www.wise.virginia.edu/philosophy/phil206/Leibniz.html ↩

1-3. Version 1 of PowerShell shipped in 2006, and provided the implementation for these cmdlets. Cmdlets today are written in .NET languages, and consist of a single class per cmdlet. PowerShell provides a base class that does much of the heavy lifting; developers define properties of the class that become parameters, and override specific methods to participate in the pipeline lifecycle. Cmdlets, along with the overall environment, were the first of four major vision points proposed in the Manifesto. ↩

1-4. Remoting was introduced in PowerShell version 2, which shipped in the box with Windows Vista and Windows Server 2008. Remoting is the second of the four major vision points proposed in the Manifesto. ↩

1-5. Although never exposed as an MMC per se, PowerShell's engine was implemented as a .NET class. Any .NET application can instantiate the engine, run commands, and translate the output into a GUI display. Exchange Server 2007 was the first product to do so, and remains one of the best examples of the "full-on PowerShell approach" to administration. ↩

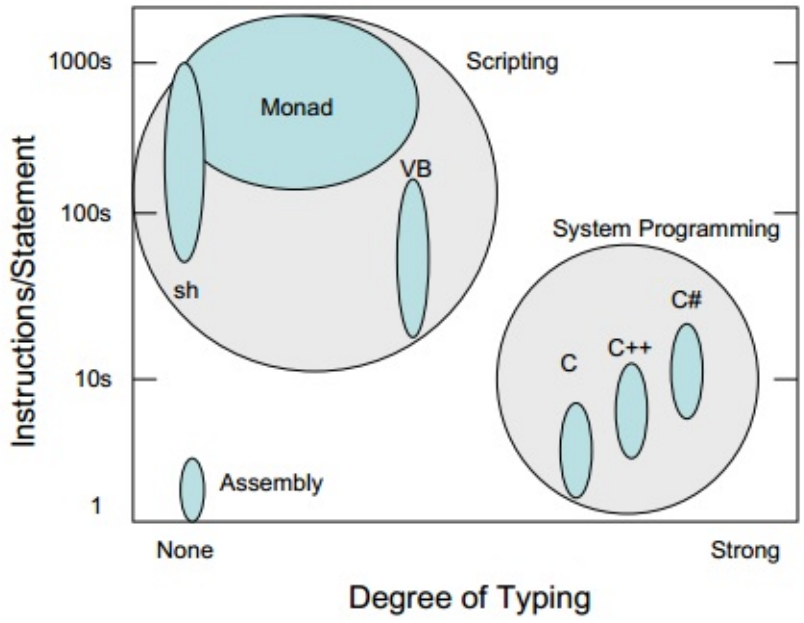
1-6. It should be noted that PowerShell very nearly didn't exist because of its dependency on .NET. At the time, in 2004-2006, a startling number of high-profile managed code projects were failing, contributing to the delays in Windows Vista. Running around Microsoft preaching about some management scripting language written in .NET wasn't politically correct at the time. In fact, it was so risky that the Exchange Server team actually built in *entire intermediate API* under their PowerShell cmdlets, on the theory that they could trash the cmdlets and switch to something else more easily, if needed. ↩

Chapter 2 - Problem

Windows has simple GUI administrative tools for basic users (Control Panel, MMC, etc). Windows also has a rich set of languages, APIs[2-1] and object models for advanced systems programmers (C, C++, C#, WMI, Win32, .Net, etc). What is missing is the vital middle – administrator-oriented composable tools to type commands and automate management. The vital middle is typically addressed by scripting languages.

Our current scripting solutions (WSH, VB) focus on the high end of the scripting world which manage the platform using very low level abstractions such as complex object models, schema, and APIs[2-2]. This is effectively systems programming and misses much of the admin community. Admin scripting flows from command line administration[2-3], it must be small, simple, incremental, and deal with very high levels of abstraction.

John Ousterhout described the distinction between scripting and systems programming well in his paper Scripting: Higher Level Programming for the 21st Century.



Ousterhout posits that scripting allows for “gluing” applications together – a higher level abstraction than system programming – enabling (even) more rapid application development than today’s systems programming languages. The fundamental argument is that we should continue to ride Moore’s Law to move development to higher levels of abstraction via script. To enable administration automation in the mainstream, administrators need a comprehensive and scriptable shell and utilities and the administrative GUIs need to be

layered on top of this infrastructure[2-4]. This will enable efficient training of administrators on command line automation, ensure comprehensive administrative capabilities at the command line, and the economies of scale of an admin-composable automation model.

Notes

> 2-1. APIs in fact, are the main differentiator between Windows and Linux/UNIX systems. On Linux/UNIX, everything essentially looks like a folder or a file, and nearly every bit of configuration is in a loosely-structured text file. Automating administration in that environment is easy, because you only have one API: text files. Windows is harder because to do anything, you've got to learn that something's API - and all the APIs are different. Knowing how to add a user to Active Directory doesn't help you create a site in SharePoint - they're all different APIs. ↩

> 2-2. Misses, in other words, the point, because VBScript is basically a simplified way of dealing with APIs that were meant for developers. VBScript also assumes that product teams have created dedicated, VBScript-compatible APIs, which most didn't. Getting anything done with VBScript was often complicated, and always hit-or-miss. ↩

> 2-3. (ORIGINAL) Administrative scripting is often the progression from ad hoc scripts to automated operations. Admins notice that they type the same commands over and over again so they build a script. The notice that their scripts contain lots of the same things so they produce parameterized subroutines and progress from there. ↩

2-4 Snover felt strongly about layering GUIs on top of command-line. That's in part because it's how many Linux/UNIX administrative GUIs do things, but it's mostly because doing it that way forces you to ensure that everything *can be done from the command-line*. The GUI doesn't become a special class of citizen holding special, unique powers; it's just another consumer of the command-line. The command-line, in turn, can be much more easily consumed by other consumers than a GUI could be.

Chapter 3 - The Traditional Approach to Administrative Automation

The traditional[3-1] model for administrative automation is powerful and successful. It consists of:

1. A programmatic shell (e.g. sh, csh, ksh, bash)[3-5]
2. A set of administrative commands (e.g. ifconfig, ps, chmod, kill)
3. A set of text manipulation utilities (e.g. awk, grep, sed).
4. Administrative GUIs layered on top of commands and utilities

This model's philosophy is that every executable should do a narrow set of functions and complex functions should be composed by pipelining or sequencing executables together. This model has been extremely successful despite serious drawbacks. Upon inspection, what is widely considered a UNIX stronghold is in fact a flawed implementation of this model[3-2].

When you step back and examine what is really going on when someone uses a pipelined command like "$ a | b | c", you conclude that the first command "a" did not accomplish what the admin wanted to do. If it had, the admin would have just type "a" and been done with it. So then the question is why didn't "a" do what the admin wanted? The answer is that in this traditional model, the stand-alone executables tightly bind three operations together: 1) getting objects; 2) processing objects; 3) outputting results as text[3-4]. One of those operations does not do what the admin needs so the rest of the pipeline is an attempt to fix that.

Because the executable outputs text, the downstream elements must use text manipulation utilities to try to get back to the original objects to do additional work. While the basic model is extremely powerful, its intrinsic flaw is the tight binding of these operations and the use of unstructured text for integration[3-3]. This requires clumsy, lossy, imprecise text manipulation utilities.

The traditional model reflects the state of the technology that was available at the time it emerged. .Net provides[3-6] a new set of capabilities and opens up the possibility of new approaches. These new approaches allow us to replace the traditional model with a decisively superior one. That model is Monad.

Notes

3-1. Traditional in the Linux/Unix world; certainly not in Windows. This is in fact the change Snover was proposing: to make administrative administration work more like it does in Unix, since Unix is a decades-proven model for success. It probably didn't hurt that Snover came from Digital Computer, a company with more than a passing familiarity with Unix variants and similar operating systems. ↩

3-2. People who view PowerShell as a "linux-ification" of Windows should note that Snover wasn't *enamored* of the Unix command-line model. He felt it was inconsistent (and, having grown organically, it is) and often lacked good semantics. In many ways, PowerShell was the first "second comer" to Unix's command-line model, taking its strengths but re-thinking what had become somewhat obvious weaknesses. ↩

3-3. There's an enormous point here that's often missed. When you write a tool that produces text, downstream tools have to know how to process that text in the exact format you produced it. Your data is unstructured. If you change the output of your tool, everything that used to work with it, won't. Object orientation - that is, presenting data in a standardized structure that could be consumed by anything understanding "objects" - was one of the biggest differences between PowerShell and what had come before. Much of a Linux admin's time is spent in the grep/sed/awk cycle, since they've got to parse out text so the next tool has data to work with; PowerShell all but eliminates that entirely ancillary work. ↩

3-4. Practical upshot of this is that tools - cmdlets, in the PowerShell world - should do one thing, and one thing only. Get objects, process objects, or format objects into text - pick just one, and do only that. If you do more than one, you start creating a monolithic tool that's less easy to re-use elsewhere. This do-one-thing concept has become a driving foundation for best practices in the PowerShell community, especially around toolmaking. ↩

3-5. These examples emphasize the influence mainframe and UNIX had on Snover's design choices. ↩

3-6. Realistically, COM could have provided the same capabilities as it was object-oriented. However, by the time the Manifesto was written, COM was effectively deprecated and Microsoft had moved on to .NET. ↩

Chapter 4 - New Approaches

Monad takes new approaches to the issues of 1) building commands, 2) composing solutions 3) management models and 4) management GUIs. The Monad architecture flows from the following observations:

1. Most solutions are home brewed and composed out of existing commands by administrators.
2. Most solutions are focused on either automating management or providing *ad hoc* fixes.
3. Most administrators are para-programmers. They either don't have the desire, skill or (more often), the time to do sophisticated programming.
4. Most application developers won't make their code manageable unless there is immediate and substantial user benefit[4-5].

4.1 - A New Approach to Building Commands

The traditional approach to building commands is inefficient. Much of the effort is spent rewriting the same functions over and over again by different people in different ways. They all:

- Parse, validate, and encode user input.
- Document usage.
- Log activity.
- Format data, output results and report errors.
- Operate on remote nodes or sets of remote nodes.

Yet, despite all this commonality, most platforms[4-14-2] provide little to no support for doing these activities in common consistent ways. The result is that today's commands are inefficient to develop and inconsistent to use[4-6].

Monad takes a different approach providing developers maximal leverage and end users maximal consistency by defining an **automation model** for applications which factors out common functions so they can be implemented once in a common runtime environment[4-3]. Developers no longer produce stand alone executables. Instead, they write narrowly focused .Net classes (**Cmdlets**) which then are exposed as APIs, commands, and GUIs. The common functions are implemented and tested once and provide a single set of semantics as well as a consistent and uniform set of error messages.[4-7]

4.2 - A New Approach to Composing Solutions

The traditional approach to composing solutions is difficult and fragile. It uses pipelines to perform prayer-based parsing of text streams[4-4]. These mechanisms are awkward, inconsistent, and imprecise. Admins spend the majority of their thought process on mechanisms instead of problem solving. Monad takes a different approach providing a precise, powerful **script execution engine** for creating pipelines of .Net objects. Instead of piping unstructured text, we pipe .Net objects[4-8]. This allows the downstream pipeline components to operate directly on the objects and their properties using the .Net Reflection APIs. (The reflection APIs allow a utility to find the type of an object, what properties/methods it has, get its property values and invoke its methods)

The Monad Runtime environment provides a means to access Cmdlets and run scripts on remote machines via Web Services.[4-9]

4.3 - A New Approach to Management Models

The traditional approach to management models produces an inconsistent admin experience. Today there are thousands of locally optimized commands. Each command developer defines his own management model with a set of names, and concepts. While copying of popular commands occurs, there is no systemic incentive for doing so. Efforts have been made to provide guidelines which would drive global optimization but the weight of legacy has made it difficult for such efforts to gain much traction.

A similar situation exists with today's instrumentation technologies which languish due to lack of tool support. Instrumentation evangelization efforts are difficult as [product] groups reject the "build it and they will come" strategy. Tool developers balk at the vast surface area of objects and respond by either providing generic functionality (like monitoring or browsing) across a broad range of objects or providing rich features for a narrow set of objects.

Monad takes a different approach: it minimizes the cost of automation and provides immediate end-user benefit by providing **scenario-based automation** extension classes and in-the-box tools that exploit those classes. Monad can support almost any automation schema but strongly encourages the use of standard schemas by providing a set of base classes for specific administrative scenarios. Those base classes include: Navigation, Diagnostics, Configuration, Lifecycle, and Operations[4-10]. These classes provide common syntax, switches, internationalized error messages and solutions to common scenario problems (e.g. a common implementation of a directory stack for all the navigation commands). Monad also provides a set of UI controls and tools that ship with the OS that drive those extensions to perform a particular management task.

4.4 - A New Approach to Management GUI Tools

The traditional approach to management GUIs provides minimal developer leverage. Today's Windows management GUI tools are developed in the same way that a full blown application is. They have GUI code, domain logic/constraint enforcement, and API access to local and remote managed objects. Management GUI services are largely limited to a UI container which facilitates multiplexing multiple tools and a certain level of integration. This approach requires a sophisticated developer and an exhaustive test matrix. Because much of the domain logic and constraint enforcement is embedded into the GUI, it is common for the command lines to expose a subset of the functions of a GUI. The traditional approach works against automation.

Monad takes a different approach providing a **rich set of management oriented services** for developing management GUI tools. These services allow management GUIs to be layered on top of the scripting engine and Cmdlets. This provides auditing, macro record/playback and integrated GUI/command line tools. This decreases the skill level required to develop a management GUI by simplifying both the access to and control of management objects and by providing transparent remoting for free. It also allows users to see the scripts run by GUI interactions which helps them learn the automation layer and create their own automated scripts. The layering reduces the test matrix by leveraging the testing done on the command line and scripts and only needing to test the GUI paths to invoke those functions. The management GUI can also expose its inner workings via Cmdlets which provides developers, testers, and support easy access to the internal state and control of the GUI for debugging/diagnostics/automated test.

Notes: [4-1]: ORIGINAL: UNIX has the getopt() call for simple command option parsing.

> [4-2]. ORIGINAL: VMS DCL and AS400's CL are the exceptions to this. They provide a common command parser so the commands that use this have a high degree of syntactic consistency. ↩

> [4-3]. ORIGINAL: There is a wonderful synergy between programmer's desire to minimize the amount of code they write for management and customers desire to have a consistent management experience. ↩

> [4-4]. Prayer based parsing is when you parse the text and pray that you got it right. e.g. Cut off the first 3 (or was it 4?) lines, cut out column 30-40 (assuming that those spaces are not tabs), cast that as an integer (hmm. – does anyone use 64 bits?...well let's just hope its 32 bits). ↩

4-5. Meaning, most developers won't implement interfaces that administrators can use to manage the application. At best, a "lazy" developer might simply put all their configuration information into a text file and call that "manageable." Ironically, that's essentially how Unix is built from the ground up, and it *is* manageable, because there's little as easy as modifying a text file, especially if it's structured (as in JSON or XML). ↩

4-6. Which is why developers hate making them and admins hate using them. ↩

4-7. This is the model PowerShell adopted. Cmdlets are instances of a class, which they inherit as their base. That class provides a ton of common functionality, so that the actual code in a cmdlet is around 99% focused on whatever it is that cmdlet is doing. The cmdlet developer doesn't focus on parsing command-line arguments, validating mandatory items, etc. ↩

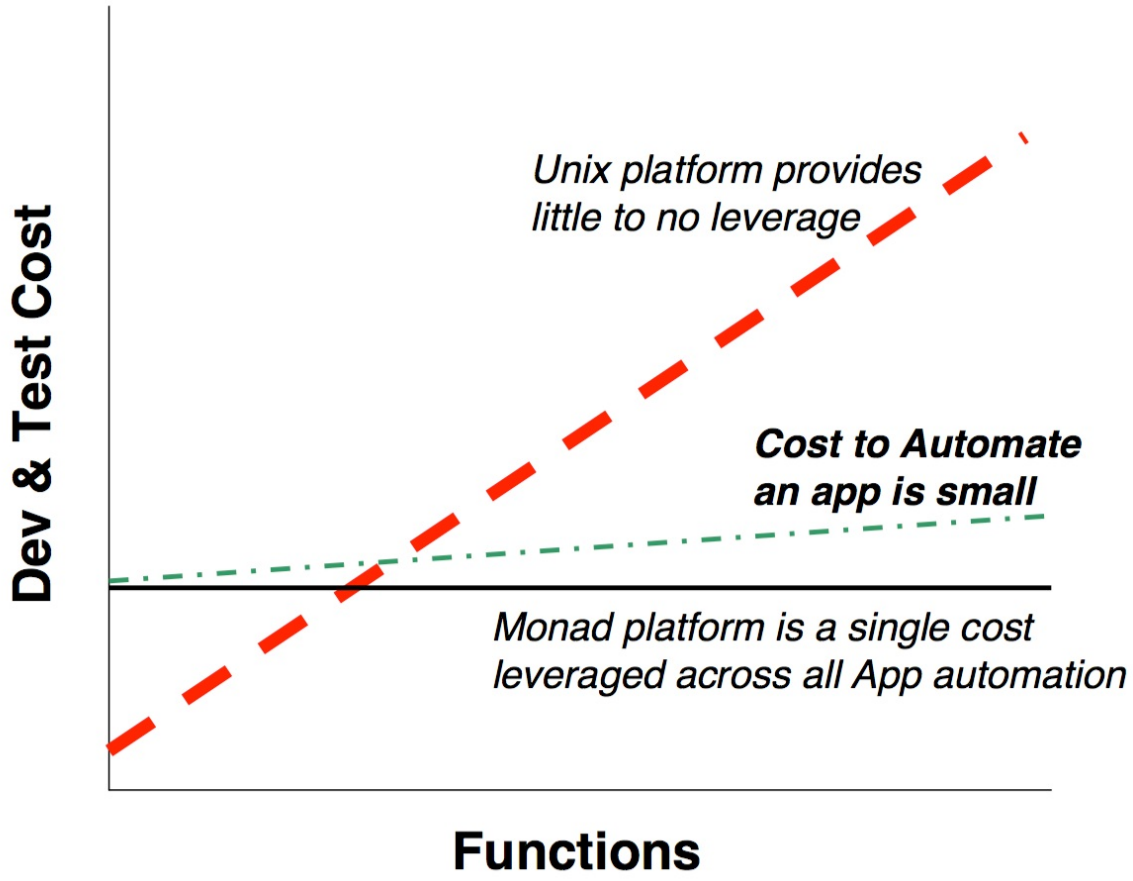
4-8. An "object" in this sense is little more than a set of structured data, not unlike a database table or a spreadsheet. Each object represents some management component, and its properties represent bits of information about that object. Commands don't have to parse these objects to find data, because .Net understands the object structure and can simply retrieve bits of information by referring to the property names. ↩

4-9. One of the first oblique references to what became PowerShell Remoting, which is indeed a web service based on WS-MAN (Web Services for Management). ↩

4-10. PowerShell never really went with specific base classes for these different scenarios, but this is the origin of PowerShell's standardized list of verbs to be used in cmdlet names. This concept also drove the creation of the PSProvider and PSDrive abstraction, wherein any data store could be exposed as a "disk drive," thus enabling a standardized set of commands to manipulate any data store so exposed. ↩

Chapter 5 - The Monad Automation Model (MAM)

Monad defines a highly leveraged automation model for applications. The model factors out common functions so they can be implemented once in the runtime environment. This provides both leverage for the developer and consistency for the administrators. The incremental cost to develop and test application-specific functions is quite low compared to the traditional methods.



Developers express an automation model to Admins as a set of user-friendly nouns and verbs. The developer implements these by subclassing a set of base automation .Net classes and annotating them with automation attributes to produce a set of Cmdlets. The MSH engine exposes these Cmdlets as APIs and a set of commands. Administrators and tool developers now get a mainstream way to uniformly access the automation of every aspect of the operating system.

5.1 - An Example

Imagine the developer who needs to expose the Windows eventlog for reporting automation. The developer decides how to structure the automation in terms of nouns and verbs ("Get-EventLog"). Monad provides strong guidance on this subject. The developer then writes a CmdLet (in C#, VB.NET, COBOL, etc) to expose this function.

A CmdLet might look like this[5-1]:

```
[CmdLet("Get", "EventLog")]
public class EventLogCmdLet : Cmdlet
{  [Parameter(Position=0)]
   public string LogName = "system"; //Default to the system log

   Protected overrride void ProcessRecord()
   {    WriteObject( new EventLog(LogName).Entries);
   }
}
```

At first glance it might appear that the Admin is not going to get much use from this code but nothing could be further from the truth. Using the CmdNoun and CmdVerb attributes automatically registers this CmdLet as the command *"Get-EventLog"* with a single parameter *"LogName"*. The Admin then uses this command along with a set of base utility commands to compose a rich set of scenarios:

What is filling up my application log?[5-2]

```
$ Get-EventLog application |Group source |Select -first 5 |Format-Table[8]
counter Property
======  ===========
  1,269 crypt32
  1,234 MsiInstaller
  1,062 Ci
    280 Userenv
    278 SceCli
```

Why is MsiInstaller filling up my log?

```
$ Get-EventLog application |Where {$_.source -eq "MsiInstaller"} `
   |Group Message |Select -first 5 |Format-Table
counter Message
======= ================================================================…
    344 Detection of product '{90600409-6E45-45CA-BFCF-C1E1BEF5B3F7}…
    344 Detection of product '{90600409-6E45-45CA-BFCF-C1E1BEF5B3F7}…
    336 Product: Visual Studio.NET 7.0 Enterprise - English - Inter…
    145 Failed to connect to server. Error: 0x800401F0
      8 Product: Microsoft Office XP Professional with FrontPage --…
```

By changing the last CmdLet in the pipeline, this information can be output in XML, CSV, LIST, HTML, EXCEL or any other format.

Is my eventlog usage regular across the week?

```
$ Get-EventLog application |Group {$_.Timewritten.DayOfWeek}
counter DayofWeek
======  ========
  1,333 Tuesday
  1,251 Wednesday
    744 Thursday
    680 Monday
    651 Friday
    556 Sunday
    426 Saturday
```

The admin can add additional Cmdlets to the pipeline to filter out only those events that where generated on Tuesday and then find out which events occur most on that day `($ Get-EventLog application |Where {$\_.TimeWritten.DayofWeek -eq "Tuesday"} |Group EventID)`. Having found that the most frequent event on Tuesdays, they can easy filter the log for that event and determine the distribution of that event across the days of the week. `($ Get-EventLog application |Where {$\_.EventID -eq 131080} |Group {$\_.TimeWritten.DayofWeek})`

Monad requires a small amount of CmdLet[5-3] code to be integrated into the runtime environment and take advantage of its rich set of functions and utilities to provide a powerful and relevant set of administrative functions. While this example focused on an *ad hoc* investigation, it is obvious how this investigation could lead to a set of automated nightly reports. This example is a narrow scenario; comprehensive Cmdlets would need to provide a full range of verbs, have the input extensively checked, and perform error handling. Still, the savings in development and test are dramatic.

5.2 - Leveraging .Net

Developers use .Net attributes to offload work to the runtime environment[5-4]. The general philosophy of Monad is to implement things once and then use them everywhere. A rich set of declarative attributes direct the Monad runtime to perform actions on behalf of the developer. This transfers the responsibility for writing and testing this code as well as for interacting with the user during error conditions and producing and localizing error messages.

Monad defines automation attributes in the following areas:

Parsing Guidance	These tell the parser how to map user input to the CmdLet Request Object. E.g. how to map parameters to properties, or whether a qualifier is mandatory.
Data Generation	These tell the new shell to process the user input to generate the actual data. E.g. filename globbing. There will also be globbers for hostnames, ipaddrs, registrykeynames, ProcessNames, etc.
Data Validation	These express validation rules on the input data. E.g. cardinality of the data, the min/max values of the data, etc.
Encoding Directives	These convey how to encode the processed user input into data objects. E.g. a CmdLet may want an array of StreamWriters instead of an array of filenames.
Object Processing	Perform a set of common functions on common datatypes. E.g. perform a ToLower() on strings.
Visibility/Applicability	These provide predicates for visiblity/applicablity. E.g. Cmdlets can be tagged with the Machine and User Roles. If a machine does not have the DHCP Server Role, the DHCP server commands will not be visible by default.
Documentation	These provide utilities information about the element. E.g. Help
Test	These provide hints to utilities to facilitate the auto generation of Test Vectors.

Notes

> 5-1. Briefly, during development, PowerShell's "script cmdlets" (now, "advanced functions") did have a syntax similar to this. In C#, cmdlet source code still looks a lot like this. ↩

> 5-2. ORIGINAL: "Get-EventLog application" is provided by the sample code above and the rest come from the Monad base commands. "Group source" counts the number of objects that have the same value for a particular property (i.e. how many times did a particular source show up?). "Select -First 5" truncates the set of objects to only have the first 5. "Format-Table" formats the objects and their properties a table ↩

> 5-3. Note that even in this document, Snover wasn't consistent about "CmdLet" versus "Cmdlet." Today, "cmdlet" is the standard. His original idea was to emphasize that a "cmdlet" wasn't a "full command" with all the parsing and whatnot a traditional command implemented; instead, it was a portion of a command, with much of the overhead being provided by the automation engine's base classes. ↩

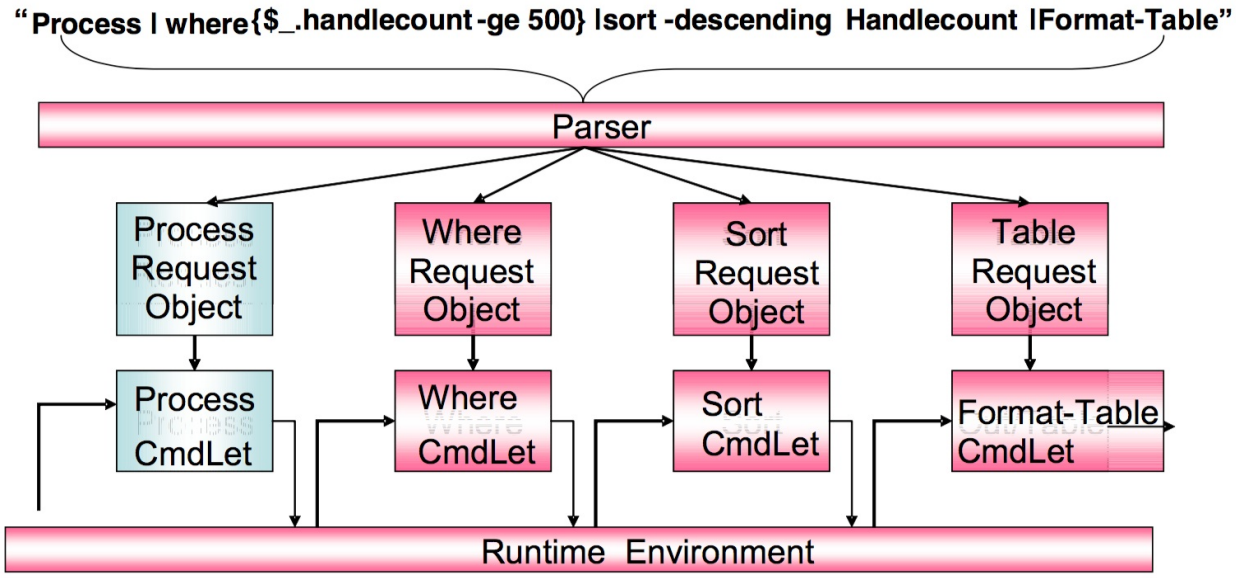
> [5-4]. Meaning, a .NET developer can tell the .NET runtime to perform certain standardized tasks. You see this a lot in PowerShell: for example, a function can declare a parameter as mandatory, and the shell will enforce that attribtue rather than the function developer having to write logic to do so. ↩

Chapter 6 - The Monad Shell (MSH)

Monad provides a runtime environment for creating highly consistent, powerful, discoverable, and secure APIs, command lines and GUIs by creating pipelines of Cmdlets. This capability is delivered as a .Net class which can be embedded in a number of "hosts" which expose this functionality to the user. The term MSH refers to both the runtime environment and the host that exposes that to the use as a command line interactive shell.

6.1 - Pipelines of .Net Objects

Monad takes user input, builds a pipeline of Cmdlets for each of the commands, parses and encodes the user input for each command into a CmdLet Request Object (CRO). The script execution engine then sequences the pipeline. The first Cmdlet is invoked and passed its CRO as a parameter. This Cmdlet returns a set of .Net objects which are then processed and passed to the next Cmdlet along with its CRO and so on until the pipeline is complete.



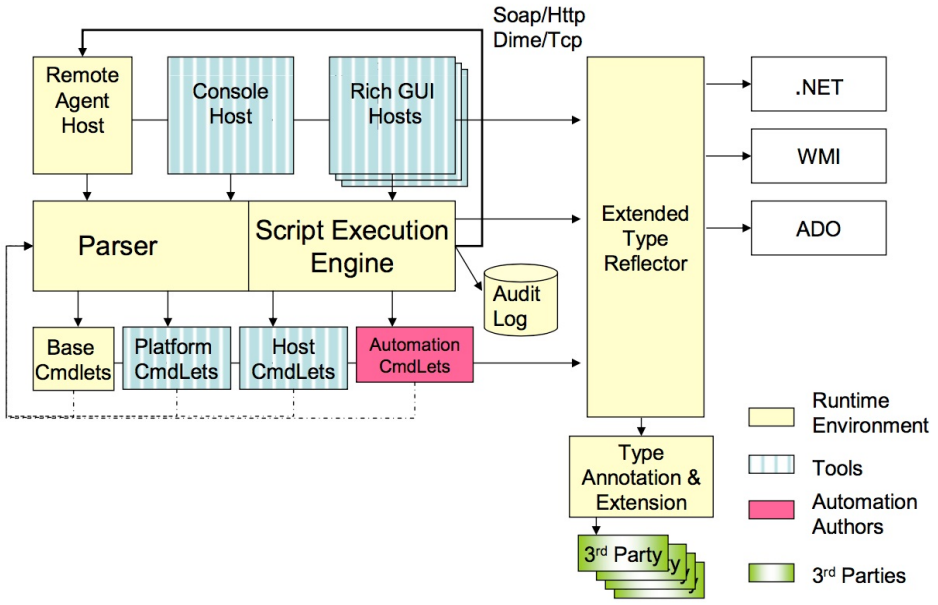
Passing .Net objects to Cmdlets instead of text streams allows reflection-based utilities to provide a function for any .Net object. In the example above, the **WHERE** CmdLet filters a set of objects based upon a test of those object's properties. It takes objects of any type (e.g. Processes, Files, Disks, etc) and queries for its type using the .Net reflection APIs. Using the Type, it queries for the existence of the property specified by the user ("HandleCount"). It uses this information to query each object for the value of that property and performs the test on that property and to filter the object appropriately.

The same mechanism is used by the **SORT** CmdLet to sort a set of objects and the **FORMAT-TABLE** CmdLet to display the properties of a set of objects as a table. Monad's utilities facilitate factoring common functions out of the Cmdlets which saves costs for the developer and increases power/consistency for Administrators.

Integrating legacy commands[6-1] is trivial because text streams are merely one type of .Net Object stream. That said, once rendered into text, you lose the ability to operate upon it as a rich reflection-based object and are back into the world of prayer based parsing.

6.2 - Monad Runtime Environment Components

The diagram below illustrates the major components of the Monad Runtime Environment:



6.2.1.1 - The Parser

The Monad parser is used by all Cmdlets and ensures a consistent syntax. It is responsible for parsing user input for the script execution engine. When a user enters a command line, the Parser maps the command to a CmdLet method and it's Request Object. The fields and attributes of the request object are used to parse the rest of the command line, generate any additional information (e.g. globbing), validate the input, and encode those values into the request object.

In performing this process, the parser augments the metadata provided by the Request Object with metadata provided by 3rd party policy providers. For instance, a request object may indicate that it can accept up to 16 nodenames and that the names must resolve to an IPv4 address. A policy can not change those directives but could add a directive indicating that the nodes must be currently responding to an ICMP ping (e.g. IsAlive).

6.2.1.2 - The Script Execution Engine

The Monad script execution engine sequences the Cmdlets and ensures a consistent runtime experience. It is responsible for taking the pipelines encoded by the parser and performing all the operations required to sequence them to completion. If the actions need to occur on a remote machine or a set of remote machines, it coordinates with the MRS. It logs all activities to the audit log. The execution engine looks at the incoming datastream and finds the correct properties to bind on a CmdLet (a CmdLet might have multiple parametersets to take advantage of different types of data). The output from a CmdLet is then gathered, potentially processed (converted, batched, etc), and passed on to the appropriate properties of the next CmdLet. Since the runtime environment can be embedded in multiple hosts (e.g. command line, GUI, etc.), it is important that a CmdLet never directly communication with the user. The script execution engine mediates this activity between the CmdLet and the various hosts.

6.2.1.3 - The Cmdlets

Cmdlets perform actions. There are four types of Cmdlets: 1) Base 2) Host 3) Platform and 4) User. **Base** Cmdlets will work in any .Net environment such as Sort, Where, Group etc. **Platform** Cmdlets are those that are dependant upon a particular platform (XP, Smart Phone, or Compact Framework) and are not available on other platforms. **Host** Cmdlets are those that are provided by the application that embeds the Monad runtime environment. For instance msh.exe, or admin GUI that expose Cmdlets specific to that host (e.g. Change a font, close a window, etc). **User** Cmdlets are those written by the User. These can be written in any language (C#, VB.NET, etc) but most will be written in MSH (the shell language).

The unique identifier for these Cmdlets is their .Net Type (e.g. System.Command.ProcessCmdLet). While this identifier can always be used to invoke the CmdLet, it is long and unfriendly. As such, CmdLet authors are required to provide Friendly names through attributes.

It will be fairly common and easy for higher order Cmdlets to be implemented by getting a set of data and then using the Monad runtime to invoke a script on that data, and then returning the results of that script.

6.2.1.4 - The Extended Type Reflector

The power of Monad is its ability to leverage .Net reflection. The problem is that there are important objects that are encoded in ways that denude reflection of its power. When you reflect against ADO datatables, you find out that they have a property called Columns. What

we need are the names of the columns but these are encoded as values. A similar problem exists with WMI, Active Directory, and XML. The extended type reflector is designed to address such issues.

6.2.1.5 - The Type Annotation and Extension System

Dealing with raw objects provides both too much and too little information. It is the job of the type annotation and extension system to resolve this paradox. It provides a mechanism for 3rd parties to define sets of properties (e.g. properties associated with performance, configuration, resource consumption, or dependencies) and give the set a public name. This allows the user to give a name instead of having to specify each and every property. E.g. "Format-Table resources" vs. "Format-Table name ,pid, workingset, handlecount, virtualmemory, privatememory".

Monad provides access to objects and the methods on those objects. However the intrinsic methods of an object represent a very small number of the interesting things that users want to do. The type extension mechanism allows 3rd parties to register brokered methods on those objects. These methods can be accessed using the same syntax as the native ones but this system will then dispatch them to the appropriate 3rd party method passing the original object as a parameter.

6.2.1.6 - The Remote Agent

Users will be able to run scripts on remote machines via Web Service requests to Remote Agent host. This host will embed the runtime and respond to requests received via Soap/HTTP or DIME/TCP. Users will be authenticated and their activities authorized (either by ID or ROLE). Requests and replies will be encoded in a way that allows cancellation and allow tracing local activities back to specific requests in remote audit logs.

When a script is complete, its return objects are serialized by value for transmission across the wire.

6.2.1.7 - Security

Monad could well be one of the most secure shell environments ever created. All interesting actions are recorded into an audit log. The code identification facilities provided by .NET significantly reduce exposure to one of the most common security exposures in a shell environment: Trojans. Signing, strong names and hashes in system policy will be used to identify which utilities are legitimate and approved and also prevent known Trojans from being executed.

In sum, the Monad shell will provide both reduced security exposures and far better detection and remediation of security breaches.

6.2.1.8 - MSH Host

MSH is a .Net assembly which can be embedded into any executable host to provide script execution and access to Cmdlets. Hosts are able to determine which subset of Cmdlets are made available to the user. The most common case will be that a Host exposes all Base Cmdlets (e.g. sort, where, etc), all of its Host Cmdlets (e.g. outlook would expose Cmdlets for dealing with mailboxes and messages), and an appropriate subset of the Platform Cmdlets (Cmdlets dealing with processes, disks, network adapters, etc).

MSH is also a stand alone executable which hosts the script execution engine and provides a rich interactive experience. While providing a compelling vt100-type experience, MSH will leverage the capabilities of a PC to provide world class analytics. MSH provides rich, graphical intellisense capabilities for command completion. Data can be output in graphical formats to leverage the PCs interaction and visualization capabilities.

6.3 - MSH Scripting Language

MSH provides a full featured scripting language using the functions and syntax of the POSIX Shell model (flow control, faulting handling, variables, function definition, scoping, IO redirection, etc) as a starting point. These are then modified and expanded upon to either improve the programming experience, take advantage of new functionality or provide a glide path to C# . The goal is that UNIX admins working with Windows will find it easy to learn and migrate their skills to MSH.

In addition to writing traditional functions, users can use the scripting capabilities of MSH to write their own Cmdlets and to add or override verbs to existing CmdLet Nouns.

Notes

> 6-1. ORIGINAL: Msh will be able to seamlessly invoke legacy commands and legacy shells will be able to seamless invoke Msh CmdLets. (Msh will provide a mechanism to export CmdLets for access from the legacy shells) [In fact, PowerShell never implemented an easy way for legacy commands to invoke cmdlets] ↩

Chapter 7 - The Monad Management Models (MMM)

Monad helps application developers design the administrative experience by providing a set of management models. A MMM is a rich set of scenario based automation base classes and a tool or set of tools that use those classes to perform a particular management scenario. These base classes cover the major management scenarios including: Navigation, Diagnostics, Configuration, Lifecycle, and Operations. The base classes provide a common way of performing these tasks across multiple resource types. This allows the admin to learn a model for managing a particular scenario and then apply that model to a wide range of problems and new situations. Developers pick the appropriate set of base classes, derive their own classes from these, and implement the appropriate methods for their resource types. The base classes provide the following:

1. A set of verbs for the scenario (e.g. Navigation has the verb set: pwd, cd, dir, pushd, popd, dirs)
2. A set of base request objects which define common qualifiers. E.g. If the scenario refers to a remote machine, the base request object would define a common qualifier - MACHINENAME. This discourages people from using the terms: NODE, SERVER, HOST, etc.
3. A set of exceptions and error messages for that scenario. E.g. There will be a standard schematized exception for “Resource unavailable” so that we don’t end up with dozens of variations [which exist today].
4. Common solutions to common scenario problems. E.g. the base classes will provide a standard solution to the problem of someone accidentally asking for too much information [get all objectsin LDAP].

Microsoft will localize all the user visible portions of these scenarios (Verbs, qualifiers, error messages, etc) so ISVs can significantly reduce their development costs by leveraging these base classes. In addition to these benefits, Monad provides UI controls to graphically display and interact with implementations of these base classes. Monad will ship with MMC plug-in tools that host these UI controls but ISVs or in-house developers can host the controls in their own management UIs. Since these controls will be accessing well defined and promulgated data and control interfaces, 3 rd parties can create replacement controls as well.

An Example

Navigation provides a example of a Management Model. There will be a base class for all Cmdlets that want to do Navigation. This will define the verbs (pwd, cd, pushd, dirs, popd, dir), common error messages, and provide common implementations for common problems (pushd, dirs, and popd will be implemented once). That base class can then be subclassed to provide a consistent admin experience for a minimal amount of code. Once the admin learns how to use this model, they will be able to use to across a wide range of resources. Navigating the filesystem will be the default case:

```
[4]$ pwd
F:\xpsh\prototype4\bin

[5]$ dir
Written                Length.kb Name
==================== ========= ================================
5/17/2002 1:02:26 PM          11 audit.txt
5/15/2002 12:56:35 PM         44 AxInterop.SHDocVw.dll
5/17/2002 12:55:28 PM         64 basecmds.dll
5/17/2002 12:55:28 PM        232 basecmds.pdb

[6]$ pushd ..

[7]$ dirs
F:\xpsh\prototype4
F:\xpsh\prototype4\bin
```

The same commands can be used to explore the Registry:

```
[2]$ pwd/reg
HKEY_LOCAL_MACHINE

[3]$ dir/reg
Name                          SubKeyCount ValueCount
============================ =========== ==========
HKEY_LOCAL_MACHINE\HARDWARE             4          0
HKEY_LOCAL_MACHINE\SAM                  1          0
HKEY_LOCAL_MACHINE\SOFTWARE            32          0
HKEY_LOCAL_MACHINE\SYSTEM               7          0

[4]$ pushd/reg HARDWARE

[5]$ dirs/reg
HKEY_LOCAL_MACHINE\HARDWARE [0x628]
HKEY_LOCAL_MACHINE
```

The same commands can be used to explore the Help system, Active Directory, SQL databases, WMI or other namespaces.

Chapter 8 - The Monad Remote Script (MRS)

Monad provides a Web Services based mechanism to execute scripts on remote systems. The scripts can be run on a single or large number (many thousands) of remote systems. The results of the scripts can be processes as each individual script completes or the results can be aggregated and processed en-masse when all have finished. A script can be executed in BestEffort or Reliable mode. BestEffort scripts are run from the existing process and if that process terminates, no effort to clean up the remote scripts is done and any outstanding results are lost. Reliable mode scripts are persisted to a local SQL store and a service handles the execution of the script. The user can log of out the machine and the service continues to process the script. The user can log back in and get the results of that job sometime in the future.

Chapter 9 - The Monad Management Console (MMC)

Monad provides a rich set of management framework service Cmdlets to facilitate to build management consoles. These services reduce development and test costs to produce admin UIs and consoles while enabling an integrated and admin experience. The services are used to produce an in-the-box management console but can also be used by third parties or in-house IT to implement their ownmanagement console. The goal is to be able to provide 50-70% of a generic management GUI tool for free just by building the right type of Cmdlets. Monad provides the following resources and services:

1. A script execution environment which provides GUIs uniform and consistent access to local and remote resources.
2. Integrated GUI and command line environment so that GUI interactions are displayed in a command line console. Users can use this to learn the automation layer and can also directly execute command line actions as well. This mechanism is also leveraged to provide macro record/playback.
3. Application-specific scripting. The application can expose its inner workings (e.g. buttons, displays, internal data structures etc) via Cmdlets to allow application specific scripting, debugging, and supportability.
4. Base UI controls associated with specific MMMs. (E.g. Navigation controls, lifecycle controls, diagnostic controls).
5. Rich set of base error messages which will be localized by MMC.
6. Declarative UI framework to allow metadata driven custom management GUIs.

Chapter 10 - Value Propositions

- For **application developers** who need to expose their administrative functions as command lines and GUIs, Monad provides a highly productive development framework.
  - Unlike building stand-alone command lines, Monad provides most of the common functions including a parser, a data validator/encoder, error reporting mechanisms, common functions like sorting/filtering/grouping/formatting/outputting and a set of management models which provide common verb sets, error messages and solutions to common problems and tools.
  - Unlike WMI/WMIC, Monad provides a simple programming model. Cmdlets are merely attributed .Net classes.
  - Unlike MMC, Monad provides strong guidance on how to perform management tasks and large benefits (reduced coding/testing) for those that follow that guidance.
- For **application testers** who want to ensure that the administrative command lines and GUIs operate correctly, Monad reduces the amount of code that needs to be tested and increases the productivity of the test process.
  - Unlike building stand-alone command lines, Monad provides a common implementation of most common functions minimizing the amount of application code to develop and test.
  - Unlike traditional management GUIs, Monad layers GUIs on top of Cmdlets so the bulk of the GUI core will already be tested when the command line is tested. Monad will also make it easier to test GUIs by exposing the inner workings of the GUI through a command line shell and by the ability to drive the GUI controls and code paths through command line scripts.
- For **power users** who want to interact with the system through command line interfaces, Monad provides a highly consistent set of commands and utilities as well as an environment that allows the creation of custom admin tools (i.e. not scenario bound).
  - Unlike cmd.exe, sh, ksh, csh, etc and traditional commands and utilities, Monad provides a common parser for all CmdLet and utilities ensuring syntactic consistency and common input error handling and messaging across all Cmdlets and utilities.
  - Unlike cmd.exe, sh, ksh, csh, etc and traditional command and utilities, Monad provides a strong prescriptive guidance and enforcement of CmdLet naming and error handling and provides a set of scenario automation base classes which make it easy and valuable for developers to follow those guidelines.
  - Unlike cmd.exe, sh, ksh, csh, etc and traditional command and utilities, Monad replaces pipelines passing text with pipelines passing .Net objects which allows

utilities to use the .Net reflection APIs to operate directly against the objects without the need to perform error-prone text parsing and object lookup.

- For **Administrators** that want to develop management scripts to automate the management of their systems, Monad provides a highly productive model for learning and effecting that automation.
  - Unlike cmd.exe, the Monad shell is based upon and extends the Bourne Shell syntax and control structures facilitating the skill transfer of Unix Admins.
  - Unlike sh, ksh, csh, etc and traditional command/utilities, Monad uses .Net objects instead of text as an integration mechanism allowing easier and more precise integration.
  - Unlike sh, ksh, csh, etc and traditional command/utilities, Monad exposes a rich error model leveraging .Net objects to expose precise details of what went wrong, where, when, and what objects where processed/unprocessed.
  - Unlike traditional management GUIs, Monad GUIs allow Admins the ability to see the inner workings of the GUI by exposing their actions via a command line console so that the Admin can learn the automation surface by using the GUI.
- For **GUI users** who want to automate their operations, Monad facilitates learning the automation layer by exposing the shell equivalents of GUI interactions.
  - Unlike traditional management GUIs, Monad GUIs are layered on top of Cmdlets so every function available in the GUI is also available via the command line. Unlike traditional management GUIs, Monad GUIs allow Admins the ability to see the inner workings of the GUI by exposing their actions via a command line console so that the Admin can see the command line equivalent of their GUI interactions.