2 Fakten zur PowerShell

2.1 Geschichte der PowerShell

In der Vergangenheit war Active Scripting manchen Administratoren zu komplex, weil es viel Wissen über objektorientiertes Programmieren und das Component Object Model (COM) voraussetzt. Die vielen Ausnahmen und Ungereimtheiten im Active Scripting erschwerten das Erlernen von Windows Script Host (WSH) und der zugehörigen Komponentenbibliotheken.

Schon im Zuge der Entwicklung des Windows Server 2003 gab Microsoft zu, dass man Unix-Administratoren zum Interview über ihr tägliches Handwerkszeug gebeten hatte. Das kurzfristige Ergebnis war eine große Menge zusätzlicher Kommandozeilenwerkzeuge. Langfristig setzt Microsoft jedoch auf eine Ablösung des DOS-ähnlichen Konsolenfensters durch eine neue Skripting-Umgebung.

Mit dem Erscheinen des .NET Frameworks im Jahre 2002 wurde lange über einen WSH.NET spekuliert. Microsoft stellte jedoch die Neuentwicklung des WSH für das .NET Framework ein, als abzusehen war, dass die Verwendung von .NET-basierten Programmiersprachen wie C# und Visual Basic .NET dem Administrator nur noch mehr Kenntnisse über objektorientierte Softwareentwicklung abverlangen würde.

Microsoft beobachtete in der Unix-Welt eine hohe Zufriedenheit mit den dortigen Kommandozeilen-Shells und entschloss sich daher, das Konzept der Unix-Shells, insbesondere das Pipelining, mit dem .NET Framework zusammenzubringen und daraus eine .NET-basierte Windows Shell zu entwickeln. Diese sollte noch einfacher als eine Unix-Shell, aber dennoch so mächtig wie das .NET Framework sein.

In einer ersten Beta-Version wurde die neue Shell schon unter dem Codenamen „Monad“ auf der Professional Developer Conference (PDC) im Oktober 2003 in Los Angeles vorgestellt. Nach den Zwischenstufen „Microsoft Shell (MSH)“ und „Microsoft Command Shell“ trägt die neue Skriptumgebung seit Mai 2006 den Namen „Windows PowerShell“.

Die PowerShell 1.0 erschien am 6.11.2006 zeitgleich mit Windows Vista, war aber dort nicht enthalten, sondern musste heruntergeladen und nachinstalliert werden.

Die PowerShell 2.0 ist zusammen mit Windows 7/Windows Server 2008 R2 erschienen am 22.7.2009.

Die PowerShell 3.0 ist zusammen mit Windows 8/Windows Server 2012 erschienen am 15.8.2012.

Die PowerShell 4.0 ist zusammen mit Windows 8.1/Windows Server 2012 R2 am 9.9.2013 erschienen.

Die PowerShell 5.0 ist als Teil von Windows 10 erschienen am 29.7.2015. Abweichend von den bisherigen Gepflogenheiten ist die PowerShell 5.0 als Erweiterung für Windows Server 2008 R2 (mit Service Pack 1) und Windows Server 2012/2012 R2 erst deutlich später am 16.12.2015 erschienen. Für Windows 7 und Windows 8.1 sollte es erst gar keine Version mehr geben. Doch am 18.12.2015 hatte Microsoft ein Einsehen mit den Kunden und lieferte die PowerShell 5.0 auch für diese Betriebssysteme nach. Kurioserweise musste Microsoft den Download dann am 23.12.2015 wegen eines gravierenden Fehlers für einige Wochen vom Netz nehmen. Microsoft hatte das Produkt im neuen Agilitätswahn nicht richtig getestet.

Windows Server 2016 (erschienen am 26.9.2016) enthält PowerShell 5.1 und Windows 10 und wurde mit dem Windows 10 Anniversary Update (Version 1607, Codename „Redstone 1“) am 2.8.2016 aktualisiert. PowerShell 5.1 ist erst seit 19.1.2017 als Add-on für Windows 7, Windows 8.1, Windows Server 2008 R2, Windows 2012 und Windows 2012 R2 verfügbar.

Eine reduzierte „Core“-Version der Windows PowerShell ist als „Windows PowerShell Core 5.1“ enthalten in Windows Nano Server, im ersten Release 2016 als Standardpaket, im zweiten in Release „1709“ als Option.

Die erste Version der plattformneutralen PowerShell Core (ohne Windows im Namen!) ist mit der Versionsnummer 6.0 am 20.01.2018 erschienen.

HINWEIS: Mit Windows 10 hat Microsoft das Auslieferungsverfahren auf „Windows as a Service“ umgestellt. Dies bedeutet, dass Microsoft über Windows Update im Sinne der neuen „agilen“ Strategie nun auch ständig neue Funktionen ausliefert. Dies betrifft ebenso die Windows PowerShell, die dann zukünftig auch auf diesem Wege häufigere Aktualisierungen erfahren kann. Wie häufig dies sein wird, ist zum Redaktionsschluss dieses Buchs noch offen.

Microsoft hat sich seit dem Jahr 2015 für andere Betriebssysteme und die Entwicklung als „Open Source Software“ (OSS) geöffnet. Dies betrifft nun auch die PowerShell: Die PowerShell Core, die am 20.1.2018 als Version 6.0 (in Nachfolge von Windows PowerShell 5.1) erschienen ist, ist OpenSource und läuft nicht nur auf Windows, sondern auch auf macOS und Linux.

2.2 Warum PowerShell einsetzen?

Falls Sie eine Motivation brauchen, sich mit der PowerShell zu beschäftigen, wird dieses Kapitel Sie Ihnen liefern. Es stellt die Lösung für eine typische Scripting-Aufgabe sowohl im „alten“ Windows Script Host (WSH) als auch in der „neuen“ PowerShell vor und Sie werden schnell erkennen, welche Vorteile Ihnen die PowerShell bietet.

Zur Motivation, sich mit der PowerShell zu beschäftigen, soll folgendes Beispiel aus der Praxis dienen. Es soll ein Inventarisierungsskript für Software erstellt werden, das die installierten MSI-Pakete mit Hilfe der Windows Management Instrumentation (WMI) von mehreren Computern ausliest und die Ergebnisse in einer CSV-Datei *(softwareinventar.csv)* zusammenfasst. Die Namen (oder IP-Adressen) der abzufragenden Computer sollen in einer Textdatei *(computernamen.txt)* stehen.

Die Lösung mit dem WSH benötigt 90 Codezeilen (inklusive Kommentare und Parametrisierungen). In der PowerShell lässt sich das Gleiche in nur 13 Zeilen ausdrücken. Wenn man auf die Kommentare und die Parametrisierung verzichtet, dann reicht sogar genau eine Zeile. Das PowerShell-Skript läuft in der Windows PowerShell und auch in der PowerShell Core unter Windows, aber nicht unter Linux und macOS, da es dort noch keine Implementierung des für den Zugriff auf die installierte Software notwendigen Web Based Enterprise Management (WBEM) und des Common Information Model (CIM) für die PowerShell gibt.

Listing 2.1 Softwareinventarisierung – Lösung 1 mit dem WSH
[3_Einsatzgebiete/Software/Software_Inventory.vbs]

```
' ----------------------------------------
' Skriptname: Software_inventar.vbs
' Autor: Dr. Holger Schwichtenberg
' ----------------------------------------
' Dieses Skript erstellt eine Liste
' der installierten Software
' ----------------------------------------
Option Explicit

' --- Einstellungen
Const Trennzeichen = ";" ' Trennzeichen für Spalten in der Ausgabedatei
Const Eingabedateiname = "computernamen.txt"
Const Ausgabedateiname = "softwareinventar.csv"
Const Bedingung = "SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'"

Dim objFSO ' Dateisystem-Objekt
Dim objTX  ' Textdatei-Objekt für die Liste der zu durchsuchenden computer
Dim i  ' Zähler für Computer
Dim computer ' Name des aktuellen computers
Dim Eingabedatei' Name und Pfad der Eingabedatei
Dim Ausgabedatei' Name und Pfad der Ausgabedatei

' --- Startmeldung
WScript.Echo "Softwareinventar.vbs"
WScript.Echo "(C) Dr. Holger Schwichtenberg, http://www.Windows-Scripting.de"

' --- Global benötigtes Objekt
Set objFSO = CreateObject("Scripting.FileSystemObject")

' --- Ermittlung der Pfade
Eingabedatei = GetCurrentPfad & "\" & Eingabedateiname
Ausgabedatei = GetCurrentPfad & "\" & Ausgabedateiname

' --- Auslesen der computerliste
Set objTX = objFSO.OpenTextFile(Eingabedatei)

' --- Meldungen
```

```
WScript.Echo "Eingabedatei: " & Eingabedatei
WScript.Echo "Ausgabedatei: " & Ausgabedatei

' --- Überschriften einfügen
Ausgabe _
"computer" & Trennzeichen & _
"Name" & Trennzeichen & _
    "Beschreibung" & Trennzeichen & _
    "Identifikationsnummer" & Trennzeichen & _
    "Installationsdatum" & Trennzeichen & _
    "Installationsverzeichnis" & Trennzeichen & _
    "Zustand der Installation" & Trennzeichen & _
    "Paketzwischenspeicher" & Trennzeichen & _
    "SKU Nummer" & Trennzeichen & _
    "Hersteller" & Trennzeichen & _
    "Version"

' --- Schleife über alle Computer
Do While Not objTX.AtEndOfStream
    computer = objTX.ReadLine
    i = i + 1
    WScript.Echo "=== Computer #" & i & ": " & computer

GetInventar computer

Loop

' --- Eingabedatei schließen
objTX.Close
' --- Abschlußmeldung
WScript.echo "Softwareinventarisierung beendet!"

' === Softwareliste für einen computer erstellen
Sub GetInventar(computer)

Dim objProduktMenge
Dim objProdukt
Dim objWMIDienst

' --- Zugriff auf WMI
Set objWMIDienst = GetObject("winmgmts:" &_
    "{impersonationLevel=impersonate}!\\" & computer &_
    "\root\cimv2")
' --- Liste anfordern
Set objProduktMenge = objWMIDienst.ExecQuery _
    (Bedingung)
' --- Liste ausgeben
WScript.Echo "Auf " & computer & " sind " & _
objProduktMenge.Count & " Produkte installiert."
For Each objProdukt In objProduktMenge
    Ausgabe _
    computer & Trennzeichen & _
    objProdukt.Name & Trennzeichen & _
    objProdukt.Description & Trennzeichen & _
    objProdukt.IdentifyingNumber & Trennzeichen & _
    objProdukt.InstallDate & Trennzeichen & _
    objProdukt.InstallLocation & Trennzeichen & _
    objProdukt.InstallState & Trennzeichen & _
```

```
    objProdukt.PackageCache & Trennzeichen & _
    objProdukt.SKUNumber & Trennzeichen & _
    objProdukt.Vendor & Trennzeichen & _
    objProdukt.Version
WScript.Echo     objProdukt.Name
Next
End Sub

' === Ausgabe
Sub Ausgabe(s)
Dim objTextFile
' Ausgabedatei öffnen
Set objTextFile = objFSO.OpenTextFile(Ausgabedatei, 8, True)
objTextFile.WriteLine s
objTextFile.Close
'WScript.Echo s
End Sub

' === Pfad ermitteln, in dem das Skript liegt
Function GetCurrentPfad
GetCurrentPfad = objFSO.GetFile (WScript.ScriptFullName).ParentFolder
End Function
```

Listing 2.2 Softwareinventarisierung – Lösung 2 als PowerShell-Skript [3_Einsatzgebiete/Software/SoftwareInventory_WMI_Script.ps1]

```
# Einstellungen
$InputFileName = "computernamen.txt"
$OutputFileName = "softwareinventar.csv"
$Query = "SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'"

# Eingabedatei auslesen
$Computers = Get-Content $InputFileName

# Schleife über alle Computer
$Software = $Computers | ForEach { Get-CimInstance -query $Query -computername $_ }
# Ausgabe in CSV
$Software | select Name, Description, IdentifyingNumber, InstallDate,
InstallLocation, InstallState, SKUNumber, Vendor, Version | export-csv
$OutputFileName -notypeinformation
```

Listing 2.3 Softwareinventarisierung – Lösung 3 als PowerShell-Pipeline-Befehl [3_Einsatzgebiete/Software/SoftwareInventory_WMI_Pipeline.ps1]

```
Get-Content "computers.txt" | ForEach {Get-CimInstance -computername $_ -query
"SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'" } | export-csv
"Softwareinventory.csv" -notypeinformation
```

2.3 Einflussfaktoren auf die Entwicklung der PowerShell

Die Windows PowerShell ist eine Symbiose aus

- dem DOS-Kommandozeilenfenster,
- den bekannten Skript- und Shell-Sprachen wie Perl, Ruby, ksh und bash,
- dem .NET Framework und
- der Windows Management Instrumentation (WMI).

Die PowerShell ist auf dem .NET Framework implementiert. Sie ist jedoch kein .NET Runtime Host mit der Möglichkeit, Befehle der Common Intermediate Language (CIL) auf der Common Language Runtime (CLR) auszuführen.

Die PowerShell verwendet ein völlig anderes Host-Konzept mit Commandlets, Objekt-Pipelines und einer neuen Sprache, die von Microsoft als PowerShell Language (PSL) bezeichnet wird. Sie ist Perl, Ruby, C# und einigen Unix-Shell-Sprachen sehr ähnlich, aber mit keiner Unix-Shell kompatibel. Nutzer der WMI Command Shell *(wmic.exe)*, die mit Windows XP eingeführt wurde, werden sich in der PowerShell schnell zurechtfinden.

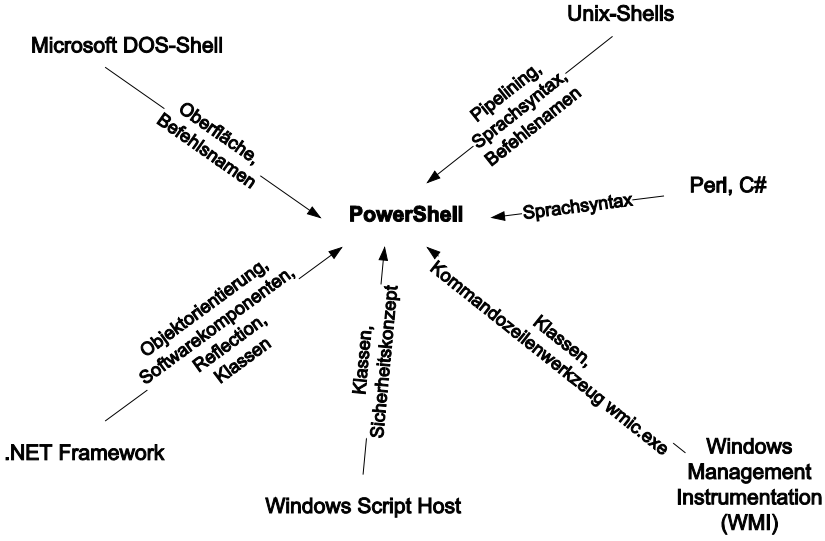

Bild 2.1 Einflussfaktoren auf die Architektur und die Umsetzung der PowerShell

ACHTUNG: Die PowerShell ist angetreten, vom Administrator weniger Kenntnisse in Objektorientierung und über Softwarekomponenten zu verlangen, als dies der Vorgänger Windows Script Host (WSH) tat. Tatsächlich kann man in der PowerShell viel erreichen, ohne sich mit dem zu Grunde liegenden .NET Framework zu beschäftigen. Dennoch: Wer alle Möglichkeiten der PowerShell nutzen will, braucht dann aber doch etwas Verständnis für objektorientiertes Programmieren und Erfahrung mit dem .NET Framework.

2.4 Betriebssysteme mit vorinstallierter PowerShell

Die folgende Tabelle zeigt, in welchen Betriebssystemen welche Version der PowerShell mitgeliefert wird bzw. wo sie nachträglich installierbar ist.

Tabelle 2.1 Verfügbarkeit der PowerShell auf verschiedenen Betriebssystemen

Betriebssystem	Mitgelieferte PowerShell	Nachträglich installierbare PowerShell
Windows 2000, Windows 9x, Windows ME, Windows NT 4.0	PowerShell nicht enthalten	Nachträgliche Installation nicht von Microsoft unterstützt
Windows XP	PowerShell nicht enthalten	PowerShell 1.0 und PowerShell 2.0
Windows Server 2003	PowerShell nicht enthalten	PowerShell 1.0 und PowerShell 2.0
Windows Server 2003 R2	PowerShell nicht enthalten	PowerShell 1.0 und PowerShell 2.0
Windows Vista	PowerShell nicht enthalten	PowerShell 1.0 und PowerShell 2.0
Windows Vista	PowerShell 1.0 enthalten	PowerShell 2.0
Windows Server 2008	PowerShell 1.0 enthalten als optionales Features	PowerShell 2.0, PowerShell 3.0
Windows Server 2008 R2	PowerShell 1.0 enthalten	PowerShell 2.0, PowerShell 3.0
Windows 7	PowerShell 2.0 enthalten	PowerShell 3.0, PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x

(Fortsetzung nächste Seite)

Tabelle 2.1 Verfügbarkeit der PowerShell auf verschiedenen Betriebssystemen *(Fortsetzung)*

Betriebssystem	Mitgelieferte PowerShell	Nachträglich installierbare PowerShell
Windows Server 2008 R2	PowerShell 2.0 enthalten	PowerShell 3.0, PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x
Windows Server 2008 Core	PowerShell nicht enthalten	PowerShell 3.0, PowerShell Core 6.x
Windows Server 2008 R2 Core	PowerShell 2.0 enthalten als optionales Feature	PowerShell Core 6.x
Windows 8.0	PowerShell 3.0 enthalten	Achtung: PowerShell 4.0 und 5.0/5.1 können nur durch ein (vorheriges) Update auf Windows 8.1 nachgerüstet werden.
Windows Server 2012 inkl. Core	PowerShell 3.0 enthalten	PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x
Windows 8.1	PowerShell 4.0 enthalten	PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x
Windows Server 2012 R2 inkl. Core	PowerShell 4.0 enthalten	PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x
Windows 10	PowerShell 5.0 enthalten	PowerShell Core 6.x
Windows 10 Creators Update (Redstone 2, Version 1703, April 2017)	PowerShell 5.1 enthalten	PowerShell Core 6.x
Windows Server 2016	PowerShell 5.1 enthalten	PowerShell Core 6.x
Windows Server 1709	PowerShell 5.1 enthalten	PowerShell Core 6.x
Windows Nano Server 2016	Reduzierte PowerShell Core 5.1 enthalten	PowerShell Core 6.x
Windows Nano Server 1709	– (PowerShell Core wurde aus dem Standardinstallationsumfang entfernt, vgl. *https://docs.microsoft.com/de-de/windows-server/get-started/nano-in-semi-annual-channel*)	PowerShell Core 5.1, PowerShell Core 6.x
Suse-Linux ab Version 42.1	-	PowerShell Core 6.x
Ubuntu-Linux ab Version 14.04	-	PowerShell Core 6.x
macOS/X ab Version 10.12	-	PowerShell Core 6.x

2.5 Anbindung an Klassenbibliotheken

Die Version 1.0 der PowerShell enthielt sehr viele Commandlets für die Pipelining-Infrastruktur, aber nur sehr wenige Befehle, die tatsächlich Bausteine des Betriebssystems in die Pipeline werfen. Prozesse, Systemdienste, Dateien, Zertifikate und Registrierungsdatenbankeinträge sind die magere Ausbeute beim ersten Blick in die Commandlet-Liste. Drei Commandlets eröffnen der PowerShell aber neue Dimensionen: `New-Object` (für .NET- und COM-Objekte) und `Get-WmiObject` bzw. `Get-CimInstance` (für WMI-Objekte). Seit Version 2.0 gibt es – zumindest in Verbindung mit neueren Betriebssystemen – mehr PowerShell-Befehle, die tatsächlich auf das Betriebssystem zugreifen.

HINWEIS: Die Option, nicht nur alle WMI-Klassen, sondern auch alle .NET-Klassen direkt benutzen zu können, ist Segen und Fluch zugleich. Ein Segen, weil dem Skriptentwickler damit mehr Möglichkeiten als jemals zuvor zur Verfügung stehen. Ein Fluch, weil nur der Skriptentwickler die PowerShell-Entwicklung richtig beherrschen kann, der auch das .NET Framework kennt. Um die Ausmaße von .NET zu beschreiben, sei die Menge der Klassen genannt. In .NET 2.0 waren es 6358, in .NET 3.5 sind es 10758, in .NET 4.7 sind es 13526.

2.6 PowerShell versus WSH

Administratoren fragen sich oft, wie sich die PowerShell im Vergleich zum Windows Script Host (WSH) positioniert, womit man neue Skripting-Projekte beginnen sollte und ob der WSH bald aus Windows verschwinden wird. Die folgende Tabelle trägt Fakten zusammen und bewertet auch die beiden Skripting-Plattformen.

Tabelle 2.2 Vergleich WSH und Windows PowerShell bzw. PowerShell Core

	Windows Script Host (WSH)	Windows PowerShell (WPS)	PowerShell Core (PS Core)
Erstmals erschienen	1998	2006	2018
Aktueller Versionsstand	5.8	5.1 und Core 5.1	6.0
Betriebssystem(e)	Alle Windows-Betriebssysteme ab Windows 95/ NT 4.0	Version 1.0 ab Windows XP, Version 5.1 ab Windows 7 und Windows Server 2008 R2; Windows PowerShell Core 5.1 auf Windows Nano Server 2016	Windows ab Version 7, Windows Server ab Version 2008 R2, diverse Linux-Distributionen, macOS

(Fortsetzung nächste Seite)

Tabelle 2.2 Vergleich WSH und Windows PowerShell bzw. PowerShell Core *(Fortsetzung)*

	Windows Script Host (WSH)	Windows PowerShell (WPS)	PowerShell Core (PS Core)
Basis-Programmierframework	Component Object Model (COM)	.NET Framework bzw. .NET Core für PowerShell Core unter Windows Nano Server 2016	.NET Core
Derzeitiger Funktionsumfang	Sehr umfangreich	Funktionsumfang in Form von Commandlets abhängig vom Betriebssystem: ▪ nur wenige Commandlets vor Windows 7, ▪ bessere Unterstützung ab Windows 7, ▪ sehr umfangreich erst ab Windows 8 bzw. Windows Server 2012. Wichtig: Auch ohne Commandlets steht auf den älteren Betriebssystemen aber ein hoher Funktionsumfang zur Verfügung, wenn man COM- oder .NET-Komponenten nutzt, was aber mehr Wissen voraussetzt.	Teilmenge von Windows PowerShell 5.1 und wenige zusätzliche neue Funktionen
Weiterentwicklung der Laufzeitumgebung	Nein, nur noch Beheben von Fehlern und Sicherheitslücken	Nein, nur noch Beheben von Fehlern und Sicherheitslücken	Ja
Weiterentwicklung der Bibliotheken	Gering, gelegentlich erscheinen noch neue COM-Bibliotheken	Ja, zahlreiche Commandlet-Erweiterungen erscheinen immer wieder mit Microsoft-Produkten.	Ja, Microsoft wird hier in den kommenden Jahren viel investieren
Weiterentwicklung der Werkzeuge	Nein	Ja	Ja
Basissyntax	Mächtig	Sehr mächtig	Sehr mächtig
Direkte Skripting-Möglichkeiten	Alle COM-Komponenten mit IDispatch-Schnittstelle einschließlich WMI	Alle .NET-Komponenten, alle COM-Komponenten, alle WMI-Klassen	Alle .NET-Standard-Komponenten. COM und WMI nur unter Windows
Skripting-Möglichkeiten über Wrapper	Alle Betriebssystemfunktionen	Alle Betriebssystemfunktionen	Viele Betriebssystemfunktionen
Werkzeuge von Microsoft	Scriptgeneratoren, Debugger, aber kein Editor	Integrated Scripting Environment (ISE), PowerShell Tools für Visual Studio, PowerShell-Erweiterung für VSCode	PowerShell-Erweiterung für VSCode, unter Windows auch ISE und PowerShell Tools für Visual Studio

	Windows Script Host (WSH)	Windows PowerShell (WPS)	PowerShell Core (PS Core)
Werkzeuge von Drittanbietern	Editoren, Debugger, Scriptgeneratoren	Editoren, Debugger, Scriptgeneratoren	Bisher nur für Windows, siehe „Windows PowerShell“
Einarbeitungsaufwand	Hoch	Mittel bis hoch (je nach Art der PowerShell-Nutzung)	Mittel bis hoch (je nach Art der PowerShell-Nutzung)
Informationsverfügbarkeit	Hoch	Mittlerweile auch sehr hoch	Für die Nutzung unter Windows sehr hoch, für die anderen Betriebssysteme noch sehr gering
Startanwendung	cscript.exe und wscript.exe	powershell.exe	pwsh.exe (Windows) bzw. pwsh (Linux und macOS)

HINWEIS: Hinweise zur Umstellung von WSH/VBScript auf die PowerShell finden Sie unter *https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-powershell-1.0/ee221101(v=msdn.10)*.