Kapitel 2

PowerShell-Formalitäten

In diesem Kapitel:

Peter Monadjemi: Windows PowerShell 2.0 - Crashkurs, Microsoft Press 2010 (ISBN 978-3-86645-667-9)

In diesem relativ kurzen Kapitel geht es um Formalitäten, die für Start und Einsatz der PowerShell 2.0 eine Rolle spielen. Dazu gehören die Installation und der Aufruf der PowerShell sowie Einstellungen, welche die Arbeitsweise der PowerShell betreffen. Im Zusammenhang mit den Profile-Skriptdateien werden bereits ein paar PowerShell-Cmdlets und Techniken vorgestellt, die erst in den folgenden Kapiteln offiziell an die Reihe kommen.

Installation und Start

Die gute Nachricht vorweg: Unter Windows Server 2008 R2 und Windows 7 muss die PowerShell nicht installiert werden, sie ist in der Version 2.0 bereits vorhanden. Bei Windows Server 2008 ist sie als Feature, das noch hinzugefügt werden muss, zwar dabei, allerdings noch in der Version 1.0. Doch auch für Anwender von Windows XP, Windows Server 2003, Vista und Windows Server 2008 gibt es eine gute Nachricht, die Installation ist unkompliziert und geht zudem relativ schnell.

Die Systemvoraussetzungen sind ebenfalls schnell beschrieben (und in den Release Notes zum Windows Management Framework enthalten): XP SP3, Vista SP1/SP2, Windows Server 2003 SP2 und Windows Server 2008 (SP1/SP2).

HINWEIS PowerShell 1.0 und 2.0 können auf einem System nicht gemeinsam installiert werden (es kann also nur eine geben).

Das Windows Management Framework

Der PowerShell-Download heißt bei Microsoft nicht *PowerShell 2.0*, sondern (neuerdings) *Windows Management Framework*. Es ist das neue Admin-Paket von Microsoft, zu dem auch die PowerShell gehört. Es umfasst neben der *PowerShell 2.0* auch *Windows Remote Management 2.0* (WinRM) und *Background Intelligent Transfer Service 4.0* (BITS) zum Übertragen von Dateien innerhalb eines Netzwerks.

Downloadformalitäten

Auch wenn es viele Wege gibt, die nach Rom, sprich zur PowerShell-Download-Seite führen, am einfachsten ist es, die allgemeine PowerShell-URL *http://www.microsoft.com/powershell* in den Browser einzugeben, die auf das Microsoft Script Center umleitet, wo wiederum, neben vielen anderen Informationen, auch die Downloadlinks angeboten werden.

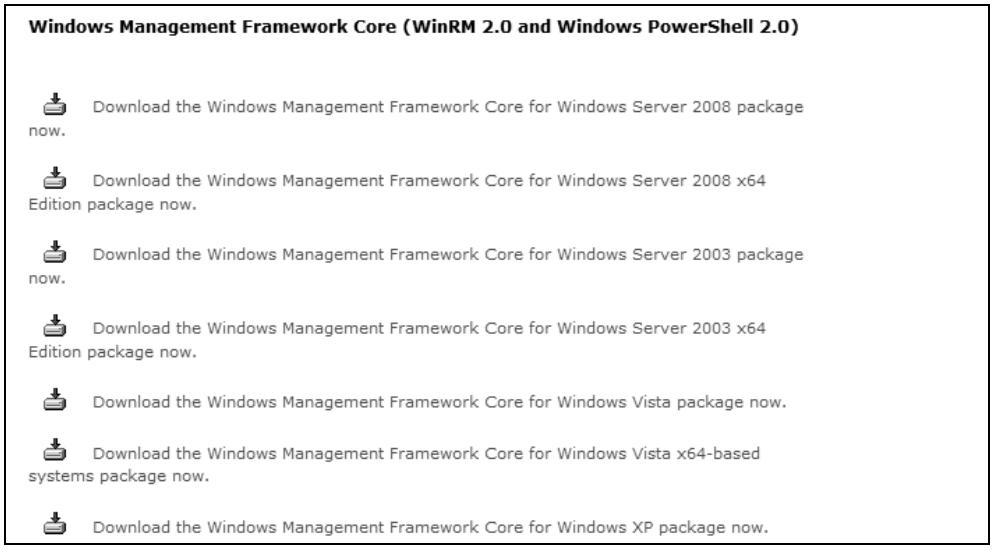

Abbildung 2.1 Auf der Download-Seite steht für jede Windows-Version und für die 32/64-Bit-Versionen eine eigene PowerShell-Version zur Auswahl (richtig übersichtlich ist die Liste nicht)

HINWEIS Unter Windows XP und Windows Server 2003 (R2) muss gegebenenfalls zuerst das .NET Framework 2.0 (genauer gesagt, die .NET Framework 2.0 Laufzeit) installiert werden, auf dem die PowerShell aufsetzt.

Nach erfolgreicher Installation wird die PowerShell über eine Verknüpfung im Startmenü aufgerufen. Sie steht gleich zwei Mal zur Verfügung: Als Windows PowerShell (*PowerShell.exe*) und als Windows PowerShell ISE (*PowerShell_ISE.exe*). Wer die PowerShell etwas einfacher starten möchte, sollte sie an das Startmenü heften oder eine Verknüpfung in der Schnellstartleiste anlegen.

HINWEIS PowerShell und PowerShell ISE sollten unter Vista und Windows 7 explizit als Administrator gestartet werden, da ansonsten für manche Operationen nicht ausreichend Berechtigungen zur Verfügung stehen.

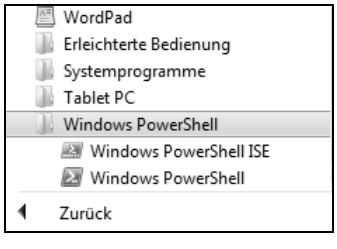

Abbildung 2.2 Unter Windows 7 steht die PowerShell von Anfang an im Startmenü zur Verfügung

Um zu erreichen, dass *PowerShell.exe* automatisch mit administrativen Berechtigungen gestartet wird, muss in den Eigenschaften der Verknüpfung in der Registerkarte *Kompatibilität* die Einstellung *Programm als ein Administrator ausführen* gesetzt sein (diese Einstellung steht nicht nur Verfügung, wenn sich die Exe-Datei in einem *System32*-Unterverzeichnis befindet, sodass die Exe-Datei dafür in ein anderes Verzeichnis kopiert werden muss).

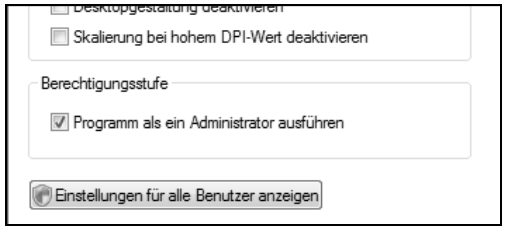

Abbildung 2.3 Diese Einstellung muss gesetzt sein, damit die PowerShell automatisch als Administrator gestartet wird

TIPP Der folgende PowerShell-Befehl liefert ein *$true*, wenn das Benutzerkonto, unter dem die PowerShell gestartet wurde, Mitglied der Administratorengruppe ist:

```
([Security.Principal.WindowsIdentity]::GetCurrent().Groups | Select-Object Value | Out-String) -match
"S-1-5-32-544"
```

In Kapitel 7 wird aus Befehlen wie diesen eine Funktion, sodass ein solcher *XXL-Befehl* etwas handlicher wird (zumal er die hoffentlich legale Abkürzung verwendet, die darin besteht, die Gruppen-SIDs mit der SID der Administratorengruppe zu »matchen«).

Das Basisverzeichnis der PowerShell 2.0

Das Basisverzeichnis der PowerShell ist *%SystemRoot%\System32\WindowsPowerShell\v1.0*. Die PowerShell 2.0 wird im selben Verzeichnis installiert wie die Version 1.0. Es kann auch über die Variable *$PSHome* abgerufen werden.

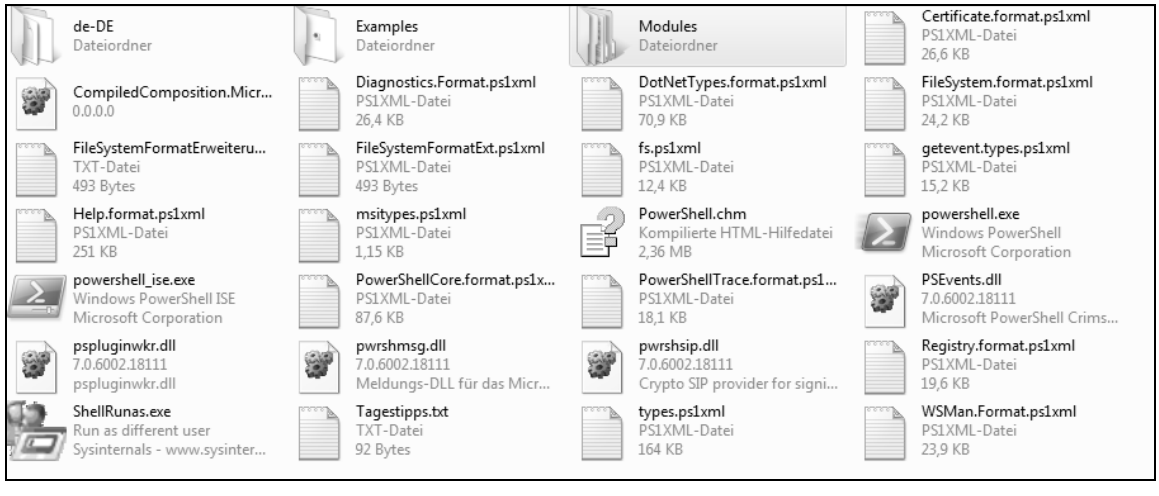

Abbildung 2.4 Der Inhalt des PowerShell-Basisverzeichnisses

Die PowerShell starten

Der PowerShell-Eingabeaufforderungshost ist eine Datei mit dem Namen *PowerShell.exe*. Wie fast jedes Befehlszeilentool besitzt sie eine Reihe von Parametern, die beim Aufruf gesetzt werden können und deren Anzahl mit der Version 2.0 deutlich erweitert wurde. Der Aufruf *PowerShell.exe -NoProfile* startet z. B. die PowerShell, ohne die (eventuell vorhandenen) Profile-Dateien auszuführen. Tabelle 2.1 stellt die wichtigsten Parameter von *PowerShell.exe* zusammen. Diese Liste erhält man auch über den Aufruf von *PowerShell.exe* mit einem »-?«.

HINWEIS Beim Aufruf von *PowerShell.exe* mit Befehlszeilenparametern kommt es auf deren Reihenfolge an. Der *File*-Parameter, der das auszuführende Skript angibt, sollte am Ende aufgeführt werden. Der folgende Aufruf startet die PowerShell und erlaubt dabei die Ausführung von Skripts, auch wenn die Ausführungsrichtlinie dies normalerweise verhindern würde:

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NoProfile -ExecutionPolicy RemoteSigned -File
C:\Pskurs\MessageBoxShow.ps1
```

Befehlszeilenparameter	Was steckt dahinter?
PSConsoleFile	Damit lässt sich eine so genannte Konsole laden, die z.B. geladene Snap-Ins umfasst und die zuvor exportiert wurde (wird relativ selten verwendet)
Version	Erlaubt es, die Version 1.0 oder 2.0 zu laden
NoLogo	Nach dem Start wird kein Copyright-Hinweis angezeigt. Es erscheint lediglich die Eingabeaufforderung.
NoExit	Die PowerShell wird nach Ausführen des angegebenen Befehls oder Skripts nicht wieder beendet (das kann nachträglich über den *Exit*-Befehl geschehen).
Sta	Startet PowerShell.exe im *Single Threaded Apartment*-Modus (und nicht im *Multi Threaded Apartment*-Modus, kurz MTA). Dieser Modus kann eine Rolle spielen, wenn ein PowerShell-Skript ein WinForms- oder WPF-Fenster anzeigen soll und dies ansonsten nicht fehlerfrei funktionieren würde (unter 1.0 war dies noch ein echter »Hack«).
NoProfile	Verhindert, dass ein Profile geladen wird (im Allgemeinen ein wichtiger Parameter)
NonInteractive	Die PowerShell wird gestartet, aber es sind keine interaktiven Eingaben möglich
InputFormat	Legt das Format für Daten fest, welche die PowerShell aus dem Standardeingabegerät lesen soll. Zur Auswahl stehen *Text* und *XML*.
OutputFormat	Legt das Ausgabeformat für den PowerShell-Output fest. Zur Auswahl stehen *Text* und *XML*. Voreingestellt ist Text. XML erlaubt manchmal eine komfortablere Weiterverarbeitung der von einer Abfrage gelieferten Daten.
EncodedCommand	Über diesen Parameter kann der PowerShell eine umfangreiche Befehlszeile zugeführt werden, in der viele Apostrophe und Anführungszeichen enthalten sind, die ansonsten »maskiert« werden müssten
File	Gibt den Pfad der Skriptdatei an, die ausgeführt werden soll (dies ist der wichtigste Parameter)
ExecutionPolicy	Legt die Ausführungsrichtlinie für die PowerShell-Sitzung fest (dies ist ebenfalls ein sehr praktischer Parameter, da unter Angabe von z.B. *RemoteSigned* ein Skript auch dann ausgeführt werden kann, wenn es die Ausführungsrichtlinie ansonsten verhindern würde).
Command	Über diesen Parameter kann ein beliebiger Befehl direkt, als Zeichenkette oder als Befehlsblock angegeben werden, der mit dem Start der PowerShell ausgeführt wird. Wird dabei ein »-«-Zeichen verwendet, wird der Befehl von der Standardeingabe gelesen. Das Resultat wird aber im XML-Format zurückgegeben, sodass sich die Objekte nicht weiterverarbeiten lassen. Falls der Befehl als Zeichenkette angegeben wird, muss der Command-Parameter der letzte Parameter sein.

Tabelle 2.1 Die wichtigsten Parameter für den Start von *PowerShell.exe*

Die PowerShell beenden

Die PowerShell wird offiziell über den *Exit*-Befehl beendet. Dieser Befehl wird in erster Linie in einem Skript benötigt, außerhalb eines solchen – bei manueller Eingabe – kann die PowerShell ja dagegen jederzeit durch das Schließen des Eingabeaufforderungsfensters beendet werden.

Kompatibilität zur Version 1.0

Skripts, die bereits mit der Version 1.0 erstellt wurden, können grundsätzlich ohne Änderungen unter der Version 2.0 ausgeführt werden. Für den relativ unwahrscheinlichen Fall, dass ein Skript eine Variable verwendet, deren Name bei der PowerShell 2.0 ein reservierter Name geworden ist, kann es natürlich zu Problemen kommen. Auch im Zusammenhang mit dem *Ieq*-Operator und dem direkten Zugriff auf die Pipeline hat es ein paar kleinere Änderungen gegeben, die in den Release Notes zum Windows Management Framework beschrieben sind.

Snap-Ins und ihre Cmdlets, die für die PowerShell 1.0 entwickelt wurden (wie z.B. die *Quest*-Cmdlets für den Active Directory-Zugriff), funktionieren genauso mit der Version 2.0.

TIPP Ein Skript kann über die Variable *$PSVersionTable* feststellen, unter welcher PowerShell-Version es läuft (bei der Version 1.0 gab es diese Variable noch nicht, sodass das Vorhandensein dieser Variablen ein Indikator dafür ist, dass das Skript von der PowerShell 2.0 oder höher ausgeführt wird).

Von Version 2.0 zurück zur Version 1.0

Auch der umgekehrte Weg kann eine Rolle spielen. Lassen sich mit PowerShell 2.0 erstellte Skripts unter einer PowerShell 1.0 ausführen? Im Prinzip ja, solange das Skript keine Elemente enthält, die es bei der Version 1.0 noch nicht gibt. Microsoft empfiehlt daher in ein Skript, das mit der Version 2.0 erstellt wurde und das auch unter der Version 1.0 ausgeführt werden könnte, zu Beginn ein *#requires -version 2.0* einzufügen. Dies hat zur Folge, dass wenn das Skript unter der Version 1.0 gestartet wird, ein entsprechender Hinweis erscheint und die Ausführung abgebrochen wird. Unter der Version 2.0 hat dieser Spezialkommentar keine Wirkung.

Die Rolle der Profile-Dateien

Die PowerShell arbeitet mit mehreren Profile-Dateien. Eine Profile-Datei ist eine reguläre Skriptdatei (Erweiterung *.Ps1*), die nach dem Start der PowerShell automatisch geladen wird. Hier werden daher Befehle eingetragen, die nach jedem Start ausgeführt werden sollen. In die Profile-Datei werden z.B. Funktionsdefinitionen aufgenommen oder eine individuelle Begrüßungsmeldung. Über das *Start-Transcript*-Cmdlet kann der *Protokollmodus* eingeschaltet werden, durch den jede Eingabe und deren Ausgaben in eine Textdatei geschrieben werden.

Die Mehrzahlform deutete es bereits an, es können mehrere Profile-Dateien Verwendung finden. Konkret sind es vier Stück, die in Tabelle 2.2 zusammengestellt sind. Es ist eine Profile-Datei für den aktuellen Benutzer und eine für alle Benutzer verfügbar. Dann gibt es eine Profile-Datei für den aktuellen Benutzer für alle PowerShell-Hosts und eine Profile-Datei für alle Benutzer für alle PowerShell-Hosts. Die Eingabe-

aufforderung, in der *PowerShell.exe* »gehostet« wird, ist nur einer von mehreren Hosts. Die *PowerShell ISE* stellt einen weiteren Host dar, ebenso z.B. *PowerGUI* von *Quest*, das in Kapitel 15 vorgestellt wird (diese Hosts können wiederum ein eigenes Paar an Profile-Dateien ausführen). Theoretisch können daher nach dem Start von *PowerShell.exe* nacheinander 4 (!) Profile-Dateien ausgeführt werden: Die beiden Profile-Dateien für den PowerShell-Host und die beiden Profile-Dateien für alle Hosts, zu denen auch der PowerShell-Host gehört.

Die Profile-Dateien werden in der Hilfe in einem Bereich beschrieben, der über *help about_profile* abgerufen wird.

Profile-Datei	Pfad
Für den aktuellen Benutzer der Microsoft PowerShell	$Env:Userprofile\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Für alle Benutzer der Microsoft PowerShell	$Env:Systemroot\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
Für den aktuellen Benutzer für alle PowerShell-Hosts	$Env:Userprofile\Documents\WindowsPowerShell\Profile.ps1
Für alle Benutzer für alle PowerShell-Hosts	$Env:Systemroot\System32\WindowsPowerShell\v1.0\Profile.ps1

Tabelle 2.2 Die verschiedenen Profile-Dateisorten und ihre Pfade

Die Variable $Profile

Der Pfad der Profile-Datei für den aktuellen Benutzer und die Windows PowerShell ist in der Variablen *$Profile* enthalten. Diese Variable besitzt eine Eigenheit, die leicht übersehen werden kann (auch wenn sie in der Hilfe erwähnt wird). Auch wenn sie für ein *String*-Objekt steht und damit eine einzelne reguläre Zeichenkette repräsentiert, wurden dem *String*-Objekt von der PowerShell vier zusätzliche *NoteProperty*-Member verliehen, über die alle vier Profile-Pfade abgerufen werden. Ein

```
PS>$Profile | Get-Member -MemberType NoteProperty

   TypeName: System.String

Name                    MemberType   Definition
----                    ----------   ----------
AllUsersAllHosts        NoteProperty System.String AllUsersAllHosts=C:\Window...
AllUsersCurrentHost     NoteProperty System.String AllUsersCurrentHost=C:\Win...
CurrentUserAllHosts     NoteProperty System.String CurrentUserAllHosts=C:\Use...
CurrentUserCurrentHost NoteProperty System.String CurrentUserCurrentHost=C:\...
```

liefert den »Beweis«, indem über das *Get-Member*-Cmdlet alle *NoteProperty*-Member mit ihrem aktuellen Wert aufgelistet werden (um den vollständigen Pfad zu sehen, muss ein | *Format-List* angehängt werden).

Der Pfad für die Profile-Datei für alle Benutzer des aktuellen Hosts ergibt sich damit über ein *$Profile.AllUsersCurrentHost*.

TIPP Dieser Tipp ist für den Anfang zwar noch viel zu speziell, aber er ist sehr lehrreich, da er deutlich macht, wie flexibel sich Informationen über die PowerShell und ihre Objekte abfragen lassen. Da ein *$Profile | Get-Member -Membertype NoteProperty* die vier *NoteProperty*-Member mit den jeweiligen Pfaden der einzelnen Profildateien auflistet, wäre das doch eine Möglichkeit festzustellen, ob die Dateien auch existieren. Der folgende Befehl macht genau das, indem er im Rahmen eines *ForEach-Object*-Cmdlets für jeden Pfad über das *Test-Path*-Cmdlet im Zusammenspiel mit dem *if*-Befehl feststellt, ob der Pfad existiert und eine entsprechende Meldung ausgibt:

```
# Alle Profile-Verzeichnisse testen
$Profile | Get-Member -Membertype NoteProperty | `
  Select-Object Definition | ForEach-Object {
  $ProfilePfad = ($_.Definition -Split "=")[1]
  if (Test-Path $ProfilePfad)
  { "Vorhanden: $ProfilePfad " }
  else
  { "Nicht vorhanden: $ProfilePfad" }
}
```

Machen Sie sich keine Gedanken, wenn alles im Moment noch ein wenig seltsam erscheint. Das wäre beim Lernen keiner der beliebten Skriptsprachen am Anfang anders. Spätestens nach Kapitel 7 dürften alle Unklarheiten beseitigt sein.

HINWEIS Die PowerShell ISE arbeitet mit ihren eigenen beiden Profile-Dateien für den aktuellen Benutzer (*$Env:UserProfile\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1*) und für alle Benutzer der ISE (*$PSHome\Microsoft.PowerShellISE_profile.ps1*).

Die Rolle der Ausführungsrichtlinie

Damit eine nicht signierte Profile-Datei nach dem Start ausgeführt werden kann, muss die Ausführungsrichtlinie der PowerShell die Ausführung von nicht signierten Skripts zulassen. Das kann über den Aufruf von

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

erreicht werden. Dieser Befehl muss nur einmalig ausgeführt werden. Er kann allerdings nicht Teil der Profile-Datei sein, da diese (natürlich) nicht ausgeführt werden kann, wenn es die Ausführungsrichtlinie nicht zulässt.

Das Laden der Profile-Datei(en) verhindern

Sollen die Profile-Dateien nach dem Start der PowerShell nicht ausgeführt werden, muss *PowerShell.exe* mit dem Befehlszeilenparameter *NoProfile* aufgerufen werden. Das ist z. B. immer dann sehr praktisch, wenn die PowerShell nur ein Skript ausführen soll oder die PowerShell innerhalb einer PowerShell-Sitzung erneut gestartet wird, aber mit einem anderen Benutzerkonto.

Eine Profile-Datei anlegen

Es ist am Anfang eventuell ein wenig irritierend, dass es weder eine Profile-Datei für den aktuellen Benutzer (und den aktuellen Host) gibt noch das Verzeichnis *WindowsPowerShell* im Benutzerprofil und damit die Variable *$Profile* auf eine nicht existierende Datei in einem nicht existierenden Verzeichnis verweist. Es ist natürlich kein Problem, das Verzeichnis anzulegen, was der folgende PowerShell-Befehl erledigt:

```
Md (Split-Path $Profile) -Force
```

Anschließend kann die Ps1-Datei mit *Notepad $Profile* angelegt werden.

Ein Beispiel für eine Profile-Datei

Auch wenn es (natürlich) keine typische Profile-Datei gibt, soll die folgende Profile-Datei ein paar Anregungen dafür geben, was eine Profile-Datei enthalten kann.

```
Start-transcript $Env:UserProfile/"PowerShellKurs.txt" -Append

# Gibt einen Tipp des Tages aus
function Get-Tagestipp
{
  if (Test-Path $PsHome\Tagestipps.txt)
  {
    $Tipps = Get-Content $PsHome\Tagestipps.txt
    $z = new-object system.random (Get-Date).Millisecond
    return $Tipps[$z.Next(0, $Tipps.Length)]
  }
}
# Prüfen, ob die Benutzerkontensteuerung aktiv ist
function Get-UACStatus
{
  $Status = (get-itemproperty -path HKLM:\Software\Microsoft\Windows\CurrentVersion\Policies\System -
Name EnableLUA).EnableLUA
  if ($Status -eq 1) { return "UAC ist aktiviert."}
  else { return "UAC ist nicht aktiviert." }
}

# Namen des aktuellen Benutzers holen
function Get-UserID
{
  return [System.Security.Principal.WindowsIdentity]::GetCurrent().Name
}

#$MaximumHistoryCount = 100
$env:Path += ";$Env:ProgramFiles\Microsoft SDKs\Windows\v6.0A\Bin"
$Host.PrivateData.ErrorBackgroundColor = "White"
write-host -fore yellow "***********************************************"
write-host -fore yellow "**            Hallo, Pemo                    **"
write-host -fore yellow "** Aktueller User: $(Get-UserID)             **"
$Tt = Get-TagesTipp
if ($Tt -ne $null)
```

```
{
   Write-Host -fore yellow "** $(New-Object System.String ' ', (24-($Tt.Length)))Tipp: $Tt$(New-Object
System.String ' ', (24-($Tt.Length)))**"
}
Write-Host -fore yellow "**$('           Es ist: {0,-10:HH:mm}' -f (get-date))              **"
Write-Host -fore yellow "**                                          **"
Write-Host -fore yellow "** Aktuelles Verzeichnis: $(gl)           **"
Write-Host -fore yellow "**                                          **"
Write-Host -fore yellow "************************************************"

# Individueller Prompt
function prompt
{
'PS ' + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}

# -------------------------------------------------------
# Hintergrund der Konsole für Admin-Mode einfärben
# -------------------------------------------------------
& {
   $WID =[Security.Principal.WindowsIdentity]::GetCurrent()
   $WinPrp = New-Object Security.Principal.WindowsPrincipal($WID)
   $Admin=[Security.Principal.WindowsBuiltInRole]::Administrator
   $IsAdmin=$WinPrp.IsInRole($Admin)
   if ($IsAdmin)
   {
    $Host.UI.RawUI.Foregroundcolor = "White"
    $Host.UI.RawUI.BackgroundColor = "Black"
    $Host.UI.RawUI.WindowTitle = "Administrator: " + $Host.UI.RawUI.WindowTitle
  }
  else
  {
    $Host.UI.RawUI.WindowTitle = "Kein Administrator !!!" + $Host.UI.RawUI.WindowTitle
  }
}

function Info()
{
  $env:UserDomain
  $env:username
  $env:ComputerName
}
```

Auch wenn das Skript recht kompliziert wirken mag, enthält es nur harmlose Befehle, die in den folgenden Kapiteln des Buchs vorgestellt werden. Es weist folgende Bereiche auf:

- Über *function*-Befehle werden eine Reihe von Funktionen definiert, mit denen sich z.B. der aktuelle Zustand der Benutzerkontensteuerung unter Vista und Windows 7 und der Name des aktuell angemeldeten Benutzers abfragen lassen. Diese Funktionen werden nicht innerhalb der Profile-Datei aufgerufen, sie stehen lediglich für den Aufruf innerhalb der PowerShell-Sitzung zur Verfügung.
- Über den Befehl *Start-Transcript* wird die Aufzeichnung gestartet, die alle Eingaben und deren Output umfasst. Wird kein Pfad angegeben, wird die Transskriptdatei unter *%userprofile%\documents* angelegt.

- Je nachdem, ob die PowerShell im Adminmodus gestartet wird, wird eine andere Hintergrund- und Vordergrundfarbe eingestellt und in der Titelleiste erscheint das Wort *Administrator* (was nicht zwingend erforderlich wäre, da dieser Hinweis, genau wie bei der PowerShell ISE auch, automatisch angezeigt wird).
- Es wird ein Banner mit einer mehrzeiligen Begrüßungsmeldung ausgegeben, die unter anderem auch einen »Tipp des Tages« enthält, der jedes Mal (per Zufallszahlengenerator) aus einer Textdatei gelesen wird, die im PowerShell-Home-Verzeichnis enthalten sein muss.
- Es wird ein neuer Prompt definiert.

Alle diese Aktivitäten sind vollkommen freiwillig und dienen in erster Linie der »individuellen Verwirklichung«.

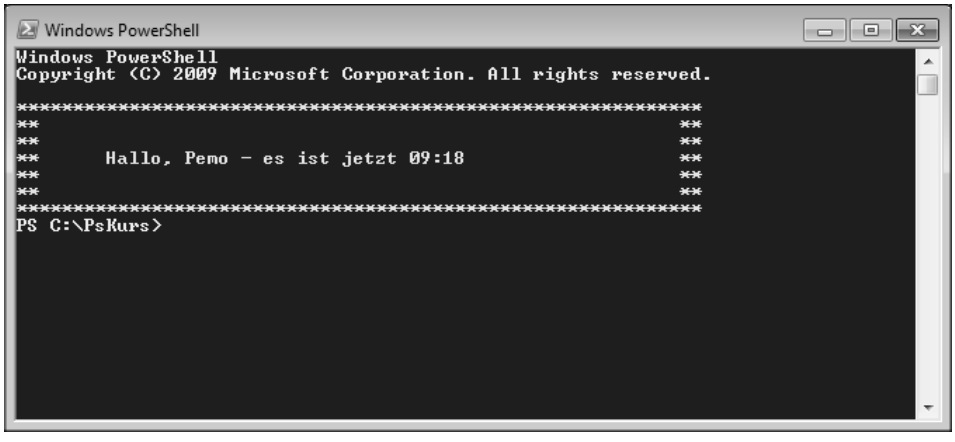

Abbildung 2.5 Die individuell eingerichtete PowerShell zeigt nach dem Start eine individuelle Begrüßungsmeldung an

Das Anlegen einer Protokolldatei (Start-Transcript)

Um die eigenen Aktivitäten mit der PowerShell später nachvollziehen zu können, gibt es die Möglichkeit, über das *Start-Transcript*-Cmdlet eine Aufzeichnung in eine Textdatei zu starten. Dabei werden nicht die ausgeführten Befehle aufgezeichnet, sondern nur die Eingaben des Benutzers und die Antwort der PowerShell (die Transcriptdatei kann daher etwas umfangreicher werden). Über das *Stop-Transcript*-Cmdlet wird die Aufzeichnung wieder beendet. Wird kein Pfad angegeben, legt die PowerShell die Datei im Benutzerprofil an und wählt einen Dateinamen, der mit *PowerShell_transcript* beginnt und auf den das Datum und die Uhrzeit des Aufzeichnungsbeginns folgen.

HINWEIS Enthält die Profile-Datei den Befehl *Start-Transcript*, kommt es bei einem weiteren Start der PowerShell aus der aktuellen Sitzung heraus mit Profile-Datei zu einer Fehlermeldung (die aber harmlos ist), da die Aufzeichnung nicht erneut gestartet werden kann, wenn sie bereits läuft.

HINWEIS Leider enthält das *Start-Transcript*-Cmdlet einen kleinen Bug. Er führt dazu, dass auch bei Verwendung des *ErrorAction*-Parameters mit einem *SilentlyContinue* ein Fehler angezeigt wird, wenn die Aufzeichnung aus irgendeinem Grund nicht gestartet werden kann. Möchte man die Fehlermeldung unterdrücken, muss über den *ErrorAction*-Parameter mit dem Wert *Stop* ein terminierender Fehler ausgelöst werden, der in einem *try-/catch*-Block abgefangen werden kann:

```
try {
 Start-Transcript -Path $TransPfad -Append -ErrorAction Stop }
catch {
 write-host -fore red "Fehler: Transcript-Aufzeichnung konnte nicht gestartet werden.`n" }
```

Das *Stop-Transcript*-Cmdlet, mit dem eine Aufzeichnung beendet wird, führt zu einem Fehler, wenn keine Aufzeichnung läuft. Da auch hier ein *-Error SilentlyContinue* scheinbar keine Wirkung hat und sich nicht feststellen lässt, ob eine Aufzeichnung läuft oder nicht, muss das obige Konstrukt verwendet werden, um eine etwas störende Fehlermeldung zu vermeiden.

Die Eingabezeile der Eingabeaufforderung

Die PowerShell 2.0 kann auf zwei Arten bedient werden: traditionell wie vermutlich jede Shell in einer Eingabeaufforderung mit Eingabeprompt und Tastatureditor. Oder ein wenig neumodisch im Rahmen einer grafischen Umgebung, der ISE, die im nächsten Abschnitt vorgestellt wird. In diesem Abschnitt werden Sie fit für die Eingabeaufforderung gemacht, die zwar konzeptionell noch aus den 1980er-Jahren stammt, mit der man aber auch im Jahre 2010 effektiv arbeiten kann.

Die Groß- und Kleinschreibung spielt keine Rolle

Die wichtigste Regel ist, dass die Groß- und Kleinschreibung bei einem PowerShell-Befehl keine Rolle spielt. Auch zusätzliche Leerzeichen innerhalb einer Befehlszeile sind bedeutungslos.

Die Eingabe ist auch über mehrere Zeilen möglich

Ein Befehl muss nicht komplett in die Befehlszeile eingegeben, sondern kann mittendrin per ↵ umgebrochen und in der nächsten Befehlszeile fortgesetzt werden. Dabei zeigt die PowerShell mit >> einen speziellen Prompt an, der immer dann erscheint, wenn die PowerShell einen Befehl nicht ausführen kann, da er noch nicht komplett eingegeben wurde. Entweder vervollständigt man dann den Befehl oder bricht die Eingabe mit einem Strg+C ab.

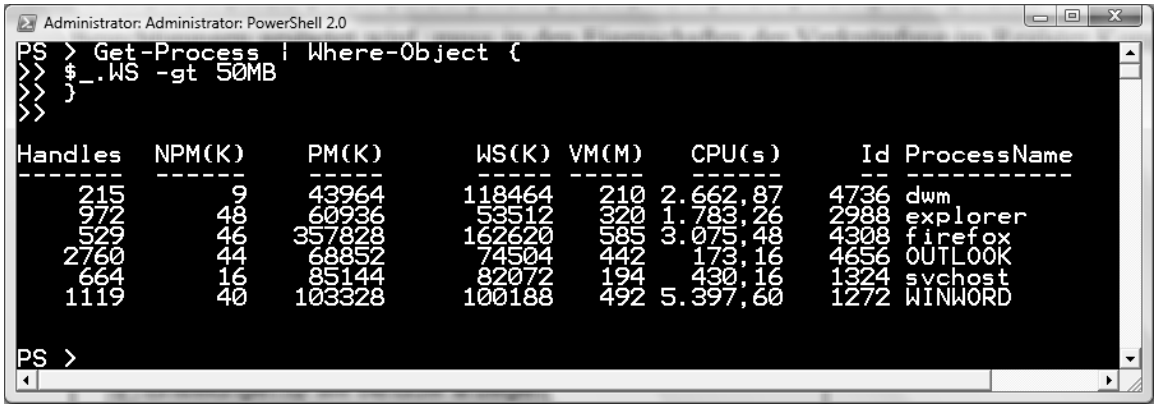

Abbildung 2.6 Die PowerShell wartet auf die Vervollständigung der Eingabe

Umbruch mit Zeilenfortführungszeichen

Ein Befehl kann nicht überall unterbrochen werden. Um der PowerShell zu signalisieren, dass die Eingabe in der nächsten Befehlszeile fortgesetzt wird, gibt es das unscheinbare Tickzeichen `, das per ⇧+´ erzeugt wird.[1]

Der Befehlszeileneditor

Die Eingabeaufforderung bietet einen einfachen Befehlszeileneditor, bei dem sich die Befehlszeile mit den Pfeiltasten editieren lässt. Außerdem können die einzelnen (durch Leerzeichen getrennten) Befehlsbestandteile über Strg+← und Strg+→ angesprungen werden, was sehr praktisch ist. Über F3 wird die zuletzt eingegebene Zeile erneut abgerufen.

TIPP Auf diese nette Einrichtung kommt man vermutlich nur per Zufall: Über ⇆ wird nicht nur erreicht, dass ein eingegebener Cmdlet-Name falls erforderlich vervollständigt (dazu später mehr), sondern auch auf Groß- und Kleinschreibung getrimmt wird, sodass Hauptwort und Verb, wie es sich gehört, groß-, der Rest des Wortes jeweils kleingeschrieben wird.

Kopieren und Einfügen

Texte aus der Zwischenablage können in die Befehlszeile mit der rechten Maustaste eingefügt werden, wenn die entsprechende Einstellung *Einfügemodus* (in den Eigenschaften der Eingabeaufforderung) aktiviert ist. Das ist besonders bei Pfaden sehr praktisch, die aus einem Textfenster oder einer anderen Shell herauskopiert werden. Umgekehrt lassen sich bei aktivierter *Quick-Edit-Modus*-Einstellung Texte im Eingabeaufforderungsfenster markieren und über ↵ in die Zwischenablage kopieren.

TIPP Über das Einfügen mit der rechten Maustaste lassen sich auch komplette Skripts, die z.B. von einer Webseite kopiert wurden, einfügen und ausführen (gegebenenfalls muss der letzte >>-Prompt durch Drücken von ↵ bestätigt werden).

[1] Und in der PowerShell-Hilfe *Gravis*-Zeichen heißt.

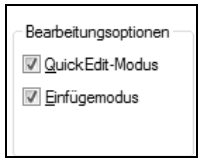

Abbildung 2.7 Diese Einstellungen müssen gesetzt sein, damit ein Kopieren und Einfügen im Eingabeaufforderungsfenster möglich wird

Das ist wichtig: Beachten Sie, dass der Fensterpuffer nicht mehr Zeichen umfassen sollte als die Fensterbreite. Ansonsten existiert ein unsichtbarer rechter Rand und Sie müssen scrollen, um die Zeile vollständig sehen zu können. Das ist generell ungünstig.

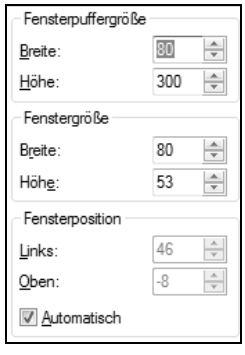

Abbildung 2.8 Der Fensterpuffer sollte nicht breiter als das Fenster sein

Eingegebene Befehle erneut abrufen

In der Eingabeaufforderung wird der Befehlszeilenpuffer über F7 abgerufen. Auf diese Weise kann eine eingegebene Befehlszeile erneut abgerufen werden. Soll eine Befehlszeile nur in die Eingabezeile geholt werden, geschieht dies mit →, soll sie dagegen gleich ausgeführt werden, drücken Sie ↵.

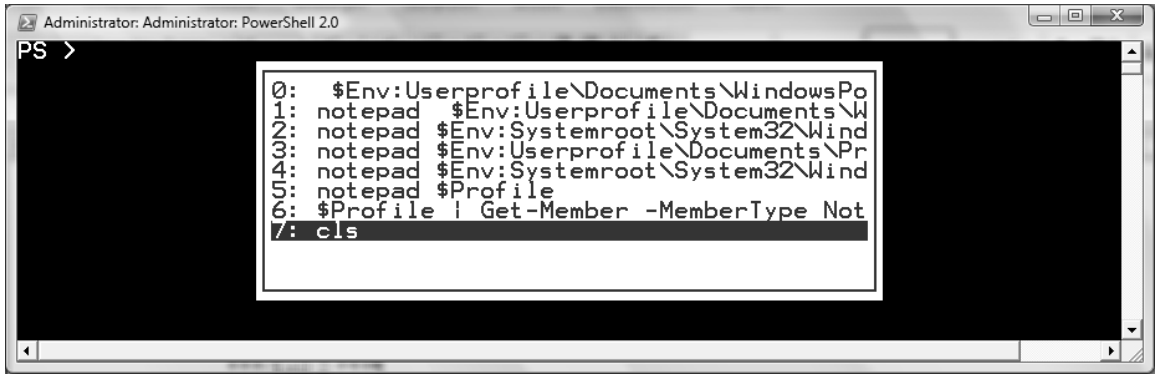

Abbildung 2.9 Über F7 wird der Befehlspuffer der Eingabeaufforderung abgerufen

Autovervollständigung mit ⇆

Das ist beinahe der wichtigste Tipp für das Arbeiten mit der PowerShell-Eingabeaufforderung. Sie müssen sowohl bei Namen von Cmdlets als auch bei Dateipfaden nur jeweils die ersten paar Zeichen eingeben und erreichen durch Drücken von ⇆, dass die Eingabe entweder vervollständigt wird oder die zur Auswahl stehenden Optionen durch erneutes Drücken von ⇆ der Reihe nach ausgewählt werden können. Das ist enorm praktisch, da längere Cmdletnamen und Dateipfade dadurch niemals vollständig eingegeben werden müssen.

Sonderzeichen und ihre Bedeutung

Bei der PowerShell spielt fast jedes Sonderzeichen eine Rolle. Während Zeichen wie der Apostroph, das Dollarzeichen, der Punkt, das Caretzeichen (^), der senkrechte Strich oder die geschweifte Klammer im EDV-Alltag ihr recht tristes Dasein als Satzzeichen fristen oder gar keine Rolle spielen, erhalten diese »Sonderlinge« bei der PowerShell eine neue Daseinsberechtigung.[2] Der Star unter den Sonderzeichen ist natürlich der senkrechte Strich, denn er steht für den Pipe-Operator, der den Outputkanal eines Cmdlets mit dem Inputkanal des folgenden Cmdlets verbindet.

Tabelle 2.3 stellt die wichtigsten Sonderzeichen zusammen, die bei der Eingabe eine Rolle spielen. Die PowerShell-Operatoren werden nicht berücksichtigt.

Sonderzeichen	Bedeutung
|	Pipe-Operator
$	Leitet eine Variable ein
.	Führt einen dotsourced-Aufruf einer Skriptdatei aus oder steht für das aktuelle Verzeichnis
..	Bereichsoperator (z.B. 1..10)
!	Not-Operator
>	Umleitungsoperator
<	Umleitungsoperator
%	Alias für das *ForEach-Object*-Cmdlet
?	Alias für das *Where-Object*-Cmdlet
@	Leitet ein Array oder eine Hashtable ein
{ und }	Leitet einen Befehlsblock oder eine Hashtable ein bzw. schließt diese(n) wieder ab
(und)	Leitet einen Ausdruck oder ein Array ein bzw. schließt diesen/dieses wieder ab
[und]	Leitet den Zugriff auf ein Element eines Arrays oder einer Hashtable ein oder schließt diesen ab
`	Zeilenfortführung (⇧ + ´)
,	Bewirkt, dass ein Array nicht Element für Element, sondern als Ganzes als Parameter übergeben wird ▶

[2] Interessanterweise sind noch nicht alle Sonderzeichen vergeben. Die Tilde (~) wartet noch auf eine neue Herausforderung.

Sonderzeichen	Bedeutung
;	Trennt zwei Befehle innerhalb derselben Zeile
#	Kommentarzeichen
::	Ermöglicht das Ansprechen eines Shared-Members eines Objekts
^	Über die Variable $^ wird der zuletzt ausgeführte Befehl abgerufen

Tabelle 2.3 Die wichtigsten Sonderzeichen bei der PowerShell und ihre Bedeutung

Escape-Zeichen

Soll ein Sonderzeichen nicht die Bedeutung besitzen, die es normalerweise aufweist, muss es »escaped« werden. Das erledigt das ` (Tickzeichen), das über ⇧+´ eingegeben wird und auch die Rolle des Zeilenfortführungszeichens spielt. Ein

```
$Zahl = 123
"Der Wert von $Zahl = $Zahl"
```

gibt *Der Wert von 123 = 123* aus. Soll die erste Variable nicht durch ihren Wert ersetzt werden, muss das $-Zeichen »espaced« werden:

```
"Der Wert von `$Zahl = $Zahl"
Der Wert von $Zahl = 123
```

Ein

```
"Zeile 1 `nZeile 2"
```

gibt *Zeile 1* und *Zeile 2* in zwei Bildschirmzeilen aus, da `n zu einem Zeilenumbruch führt. Ist dies nicht erwünscht, muss dem Tick ein weiteres Tickzeichen vorausgehen, welches das erste Tickzeichen »escaped«:

```
"Zeile 1 ``nZeile 2"
```

Variablen in einer Zeichenkette nicht austauschen

Sollen Variablen in einer Zeichenkette nicht durch ihren Wert ausgetauscht werden (man spricht auch von einer »*Erweiterung*), muss die Zeichenkette lediglich in einfache Apostrophe gesetzt werden. Das ist gleichzeitig das Unterscheidungsmerkmal zwischen Anführungszeichen (») und den einfachen Apostrophen (´).

Zeilenumbruch & Co

In eine Zeichenkette können eine Reihe von Buchstaben mit einer Sonderfunktion eingefügt werden, die z.B. für einen Zeilenumbruch stehen. Damit ein solches Spezialzeichen erkannt wird (Tabelle 2.4), muss ihm ein Tickzeichen (⇧+´) vorausgehen.

Escape-Zeichenfolge	Bedeutung
`n	Führt zu einem Zeilenumbruch.
`t	Fügt ein [⇆]-Zeichen ein.
`a	Gibt einen Signalton aus (nicht in der ISE).
`b	Fügt ein [←]-Zeichen ein (nicht in der ISE).

Tabelle 2.4 Escape-Sonderzeichen

Die Eingabe von Zahlen

Bei der Eingabe von Zahlen hält die PowerShell ein paar Besonderheiten bereit:

- Das Dezimaltrennzeichen ist der Punkt und nicht das Komma (was im Grunde selbstverständlich ist). Das Komma spielt die Rolle eines Trennzeichens zwischen mehreren Werten, die als ein (Array-)Wert behandelt werden sollen.
- Für Größenangaben in Byte stehen die bekannten Abkürzungen *KB*, *MB*, *GB* und mit der Version 2.0 sogar *TB* und *PB* bereit. Wer z. B. wissen möchte, wie viele CDs auf den neuen Terabyte-Memorystick passen, gibt einfach **1TB/700MB** ein und weiß Bescheid.
- Zahlen im hexadezimalen Format werden in der Form 0x*Zahl* eingegeben, z. B. 0xA (=10)

Individuelle Einstellungen vornehmen

Allzu viel einstellen lässt sich bei der PowerShell nicht, aber es gibt ein paar Kleinigkeiten, die konfiguriert werden können:

- Die Vorder- und Hintergrundfarbe des Host-Fensters
- Die Vorder- und Hintergrundfarbe der verschiedenen Meldungstypen
- Den Eingabeprompt neu gestalten
- Die Umgebungsvariable Path erweitern

Die Vorder- und Hintergrundfarbe des Host-Fensters

Die Vorder- und Hintergrundfarbe des Host-Fensters werden über die Properties des *Host*-Objekts eingestellt, das den gesamten Host repräsentiert. Und wo bekommt man dieses Objekt her? Ganz einfach, es wird von der *$Host*-Variablen der PowerShell geliefert. Ein

```
$Host | Get-Member
```

listet alle Member auf, was bei einem unbekannten Objekt generell eine gute Idee ist. Doch wo werden die Farben eingestellt? Das geschieht über die unscheinbare Property *UI*, die wiederum für ein Objekt steht, dessen noch etwas unscheinbare Property *RawUI* ein weiteres Objekt repräsentiert, das endlich Properties wie *ForegroundColor* und *BackgroundColor* besitzt. Der

```
$Host.UI.RawUI.BackgroundColor = "Cyan"
```

setzt die Hintergrundfarbe auf *hellblau*. Durch ein *Clear-Host* wird das Host-Fenster mit der neuen Farbe neu gestrichen, sprich gelöscht, sodass sich die Änderung sichtbar auswirkt. Als Farbwert stehen nur die 16 Grundfarben zur Auswahl (dazu ein kleiner Tipp: Durch Zuweisen eines Fantasiewertes werden in der Fehlermeldung die erlaubten Namen angezeigt).

Die Titelleiste des Fensters einstellen

Über ein *$Host.UI.RawUI.WindowTitle* erhält das Eingabeaufforderungsfenster einen neuen Titel.

Die folgende Befehlsfolge prüft zunächst, ob der aktuell angemeldete Benutzer ein Mitglied der Administratorengruppe ist, und sollte dies nicht der Fall sein, setzt sie ein *Kein Administrator!!!* vor den Fenstertitel.

```
# -------------------------------------------------------------
# Beispiel 2.1 - Feststellen, ob der aktuelle Benutzer
# Mitglied der Administratorengruppe ist
# -------------------------------------------------------------
$WID =[Security.Principal.WindowsIdentity]::GetCurrent()
$WinPrp = New-Object Security.Principal.WindowsPrincipal($WID)
$Admin=[Security.Principal.WindowsBuiltInRole]::Administrator
$IsAdmin=$WinPrp.IsInRole($Admin)
if (!$IsAdmin) {
  $Host.UI.RawUI.WindowTitle = "Kein Administrator !!!" + $Host.UI.RawUI.WindowTitle
}
```

Listing 2.1 Prüfen, ob der aktuelle Anwender Administrator ist

Auch hier gilt »Keine Panik!«, so kompliziert geht es bei der PowerShell nur in Ausnahmefällen zu. Wer diese Befehlsfolge häufiger benötigt, macht daraus eine Funktion, nennt sie *Check-Admin*, baut sie in die Profile-Datei ein, sodass sie nach jedem Start zur Verfügung steht, und hat es von da an z.B. sehr viel leichter zu prüfen, ob der angemeldete Benutzer ein Admin ist.

Die Größe des Eingabeaufforderungsfensters verändern

Theoretisch lässt sich auch die Größe des Eingabeaufforderungsfensters verändern. Theoretisch deswegen, weil sich das Fenster nur innerhalb der in den Eigenschaften der Eingabeaufforderung vermerkten Größenangabe skalieren lässt. Sie können das Fenster zunächst folglich nur verkleinern. Der folgende Befehl stellt eine neue Fenstergröße ein, die aber die eingestellte Größe nicht überschreiten darf:

```
$Host.UI.RawUI.WindowSize = New-Object -Type System.Management.Automation.Host.Size 50, 50
```

Vorder- und Hintergrundfarbe bei Meldungstypen einstellen

Das Einstellen der Vorder- und Hintergrundfarbe bei einzelnen Meldungstypen ist vor allem für die Fehlermeldung von Bedeutung, da rote Schrift auf schwarzem Hintergrund nicht immer ganz optimal lesbar ist.[3] Der Befehl

```
$Host.PrivateData.ErrorBackgroundColor = "White"
```

setzt die Hintergrundfarbe auf weiß, was den Kontrast etwas verbessert. Welche Farbeinstellungen über *###PrivateData* darüber hinaus möglich sind, verrät ein *$Host.PrivateData | Get-Member.*

Den Eingabeprompt gestalten

Der Eingabeprompt ist der Text, der in der Eingabezeile am linken Rand erscheint. Wird nichts anderes festgelegt, zeigt die PowerShell ein *PS* an, auf das der aktuelle Pfad folgt. Gerade das *PS* ist wichtig, da man an dem Kürzel auf einen Blick erkennen kann, ob die Eingabeaufforderung die PowerShell anzeigt oder *Cmd.exe* (manchmal ist es praktisch, *Cmd.exe* aus der PowerShell heraus zu starten, sodass dasselbe Fenster auf einmal eine andere Shell hostet). Ein kleiner Nachteil bei der Anzeige des aktuellen Pfades ist es, dass der Pfad oft sehr lang ist, sodass mehr als die Hälfte der Eingabezeile durch den Prompt belegt ist. Dieser Umstand, und generell der Wunsch nach einer Individualisierung, sind Gründe dafür, den Prompt zu modifizieren. Das geschieht, indem eine neue Funktion mit dem Namen *Prompt* definiert wird, in der die Ausgabe festgelegt wird. Die *Prompt*-Funktion wird wiederum in einer Profile-Datei untergebracht, damit der neue Prompt nach dem Start der PowerShell aktiv wird.

Einen »vernünftigen« Prompt zu kreieren, ist nicht ganz so einfach, wie es sich anhört. Man muss im Allgemeinen eine Weile probieren, bis der Prompt das gewünschte Aussehen besitzt. Die folgende *Prompt*-Funktion versteht sich daher nur als Empfehlung. Der neue Prompt zeigt den aktuellen Pfad in einer Zeile (in einem vornehmen Dunkelgrau) und darunter die Anzahl der bereits eingegebenen Befehle an, die über die *Id*-Property des letzten über das *Get-History*-Cmdlet abgerufenen Befehls ermittelt wird.

```
# -------------------------------------------------------------
# Beispiel 2.2 - ein alternativer Prompt
# -------------------------------------------------------------
function Prompt
{
  $Id = 1
  $HistoryItem = Get-History -Count 1
  if ($HistoryItem)
  { $Id = $HistoryItem.Id + 1 }
  Write-Host -ForeGroundColor DarkGray "`n$(Get-Location)"
  Write-Host -NoNewLine "PS:$ID>"
  $Host.UI.RawUI.WindowTitle = "$(Get-Location)"
  "`b"
}
```

Listing 2.2 Ein alternativer Prompt

[3] Insbesondere dann, wenn der Inhalt mit einem Beamer an eine Wand projiziert wird.

Die Umgebungsvariable Path erweitern

Auch das Erweitern der *Path*-Umgebungsvariablen kann eine sinnvolle Aktion in der Profile-Datei sein, da sich dadurch z. B. Tools (etwa *Installutil.exe*, das für das Registrieren von Snap-Ins benötigt wird) ohne Pfadangabe aufrufen lassen, was im Allgemeinen sehr praktisch ist. Die *Path*-Umgebungsvariable wird bei der PowerShell über das *Environment*-PSDrive-Laufwerk angesprochen. Ein

```
$Env:Path
```

gibt den aktuellen Wert der Variablen aus. Ein

```
$Env:Path += ";$([System.Runtime.InteropServices.RunTimeEnvironment]::GetRunTimeDirectory())"
```

hängt an den Wert der *Path*-Variablen den Pfad der .NET-Laufzeit an, in dem unter anderem auch *InstallUtil.exe* enthalten ist. Soll ein festes Verzeichnis angehängt werden, wird alles ein wenig einfacher, da der Verzeichnispfad dann nicht so umständlich abgerufen werden muss:

```
$Env:Path += ";C:\Tools"
```

Diese Erweiterungen gelten aber nur für den Rahmen der PowerShell-Sitzung, sie müssen daher mit jedem Start der PowerShell neu durchgeführt werden.

Hilfe

Die PowerShell ist nicht nur in fast allen Bereichen äußerst selbstauskunftsfreudig, sie umfasst auch eine umfangreiche (deutschsprachige) Hilfe, in der nicht nur alle Cmdlets mit Beispielen beschrieben werden, sondern auch allgemeine Themen wie z. B. der Umgang mit regulären Ausdrücken oder den mit Version 2.0 eingeführten Jobs oder Remote-Sessions.

Eine Kurzbeschreibung zu einem Cmdlet erhält man stets über den *?*-Parameter. Eine ausführlichere Beschreibung zu einem Cmdlet lässt sich über das *Get-Help*-Cmdlet oder die etwas praktischere *Help*-Funktion anfordern. Letztere bietet den kleinen Vorteil, dass sie die oft sehr umfangreichen Texte seitenweise ausgibt (und steht damit für ein *Get-Help <Thema> | More*).

Get-Help bietet einen *Detailed*- und einen *Full*-Parameter, die sich bezüglich ihrer Wirkung nur geringfügig unterscheiden. Während bei *Full* der komplette Hilfetext zu einem Cmdlet angezeigt wird, lässt *Detailed* den Hinweise-Bereich weg, der speziellere Hinweise enthält, die aber im Allgemeinen nicht sehr umfangreich sind, sodass man die Hilfe zu einem Cmdlet über den *Full*-Parameter abrufen sollte.

Hier ein paar Beispiele. Über ein

```
Help About*
```

erhält man eine Liste aller Hilfethemen (insgesamt 89), die praktisch alle Aspekte der PowerShell ausführlich beschreiben.

Über ein

```
Help About_Operator
```

erhält man eine Übersicht über die PowerShell-Operatoren und über

```
Help About_Remoting
```

eine erste Übersicht über die Remoting-Fähigkeiten der neuen Version.

Die Hilfetexte individuell durchsuchen

Die Hilfetexte liegen (natürlich) im Textformat und (natürlich) im XML-Format vor. Es gibt daher vielfältige Möglichkeiten, sie außerhalb von *Get-Help* zu durchsuchen. Da aber auch ein *Get-Help* Objekte über die Pipeline weitergibt, ist es erforderlich, diese über das *Out-String*-Cmdlet vor dem Durchsuchen in Text zu konvertieren.

Die folgende Funktion durchsucht die Cmdlet-Hilfetexte nach einem Stichwort.

```
# -------------------------------------------------------------
# Beispiel 2.3 - Durchsuchen der Hilfe nach einem Stichwort
# -------------------------------------------------------------
function Search-Help
($Begriff)
{
  Get-Command -CommandType Cmdlet | Where-Object {
    Get-Help -Full -Ea SilentlyContinue $_ |
    Out-String | Select-String -Pattern $Begriff }
}
```

Listing 2.3 Durchsuchen der PowerShell-Hilfe nach einem Stichwort

Aufgerufen wird diese Funktion z. B. durch ein »Search-Help -Begriff *Remoting*«, was zur Folge hat, dass die Hilfetexte aller Cmdlets nach dem Wort *Remoting* durchsucht werden und jene, in denen es enthalten ist, namentlich aufgelistet werden.

Hilfethemen drucken

Einzelne Hilfethemen lassen sich natürlich jederzeit über das universelle *Out-Printer*-Cmdlet ausdrucken[4] (sollten Sie sich nicht ganz sicher sein, ob es ein solches Cmdlet überhaupt gibt, mit der Funktion aus dem letzten Abschnitt und ihrem Aufruf mit dem Suchbegriff *Printer* oder über ein *Get-Command -noun Printer* hätten Sie es herausgefunden).

HINWEIS Die verschiedenen Hilfedateien (im RTF-Format), die bei der Version 1.0 dabei waren, gibt es bei der Version 2.0 nicht mehr. Die reguläre Hilfe ist so umfangreich und enthält mit ihren zahlreichen About-Themen auch Einführungen zu allen Kernthemen, sodass ein separates Handbuch nicht mehr benötigt wird. Für alle, die lieber alles in einer Datei zusammengefasst haben möchten, bietet Microsoft (unter *http://www.microsoft.com/downloads)* mit dem *Graphical Help File* eine *Chm*-Datei an, die alle Hilfetexte zusammenfasst.[5]

[4] Leider ohne Seitenzahlen, sodass sich jemand die Mühe machen sollte und ein Cmdlet oder eine Funktion, die bzw. das etwas mehr Komfort zu bieten hat, entwickeln sollte.

[5] Allerdings noch vom November 2007 (und damit noch auf dem Stand der ersten CTP).

Die Integrated Scripting Environment (ISE)

Mit der PowerShell 2.0 gibt es (endlich) auch von Microsoft einen kleinen Editor für PowerShell-Skripts mit dem Namen *Integrated Scripting Environment*, kurz ISE[6]). Die ISE ist, trotz einiger kleinerer Versäumnisse, von denen noch die Rede sein wird, ein nettes und praktisches Programm ohne Spezialfunktionen oder den Komfort, den andere PowerShell-Editoren bieten (oder glauben bieten zu müssen). Der wichtigste Umstand ist, dass die verschiedenen Registerkarten, die sich in der ISE anlegen lassen, nicht einfach nur Eingabebereiche sind, es sind eigene *Ausführungsbereiche* (engl. *runspaces*) für das Skript, das in das Fenster eingegeben wurde. Das bedeutet z.B., dass Variablen, die in dem Skript definiert werden, im Eingabebereich der ISE nach der Ausführung des Skripts zur Verfügung stehen.

Syntaxeinfärbung

Nett ist der Umstand, dass im Editorfenster der ISE die verschiedenen Bestandteile einer Befehlszeile bereits während der Eingabe entsprechend ihrer Bedeutung eingefärbt werden (einstellen kann man die Farbzuordnung anscheinend nicht), und dass sich die Schriftgröße über einen Regler stufenlos regeln lässt.

Integrierter Debugger

Sehr praktisch, wenngleich obligatorisch bei einem Skript-Editor, ist der integrierte Debugger, durch den sich ein Skript Befehl für Befehl ausführen lässt (mehr dazu in Kapitel 8). Etwas weniger praktisch ist die Art und Weise, wie die Inhalte von Variablen während einer Skriptunterbrechung angezeigt werden (unauffälliger geht es kaum), und dass dies stets nach einer gewissen Verzögerung geschieht. Ein wenig gewöhnungsbedürftig, aber bei einem Interpreter nicht anders machbar, ist der Umstand, dass während einer Unterbrechung keine Änderungen am Skriptcode vorgenommen werden können.

Ein- und Ausgabebereich

Die ISE arbeitet mit drei Bereichen, die sich ein- und ausblenden und in ihrer Höhe verschieben lassen:

- Dem Skriptbereich (er steht für einen eigenen Ausführungsbereich)
- Dem Ausgabebereich (er kann durch das »Wischersymbol« in der Symbolleiste gelöscht werden)
- Dem Direktbereich. Hier steht die Eingabezeile der PowerShell (für den aktuellen Ausführungsbereich) zur Verfügung, sodass in der ISE, wie in der Eingabeaufforderung, alle PowerShell-Befehle direkt eingegeben werden können.

Ein wenig seltsam ist, dass bei der ISE wichtige Funktionen fehlen. Dazu gehört der Umstand, dass die beim Verlassen des Editors geladenen Skriptdateien nicht im *Datei*-Menü angeboten werden, sodass man sie sich im ungünstigsten Fall auf der halben Festplatte zusammensuchen muss und eventuell eine Weile dafür

[6] Eine reichlich überladene Abkürzung – von Irish Stock Exchange bis Institut für Sozialethik. Irgendwie erinnert mich die Abkürzung an ISS (International Space Station), was insofern etwas unpassend ist, als dass der schlichte Editor von Weltraumtechnik nicht weiter entfernt sein könnte.

braucht, bis man die Skriptdatei wieder gefunden hat, an der man letzte Woche bis spät in die Nacht gesessen hat (auf der anderen Seite führt der Wegfall solcher Bequemlichkeitsstützen dazu, dass man sich endlich so organisiert, dass man seine Skripts jederzeit wieder finden kann).

TIPP Über F8 wird nur der aktuell im Editorfenster markierte Text ausgeführt.

Auch eine Druckfunktion wurde vergessen, sodass sich aus der ISE Skripts nicht drucken lassen. Insgesamt ist die ISE ein seltsames Tierchen, insbesondere wenn man berücksichtigt, dass es von Microsoft stammt und der Konzern (mindestens) geschätzte 2–3 Jahre Zeit hatte, sie zu entwickeln. Trotz kleinerer »Kritikpunkte« ist die ISE natürlich ein großer Fortschritt gegenüber Notepad, sodass sich nach einer kurzen Eingewöhnungsphase mit der ISE produktiv arbeiten lässt.[7] Einen zwingenden Grund, die ISE zu benutzen, gibt es allerdings nicht.[8]

HINWEIS Die ISE besitzt eine kleine »Besonderheit«, die zu Irritationen führen kann. Soll eine bereits geladene, in der Zwischenzeit aber außerhalb der ISE geänderte Datei erneut geladen werden, muss die Registerkarte für das Skript erst geschlossen werden.

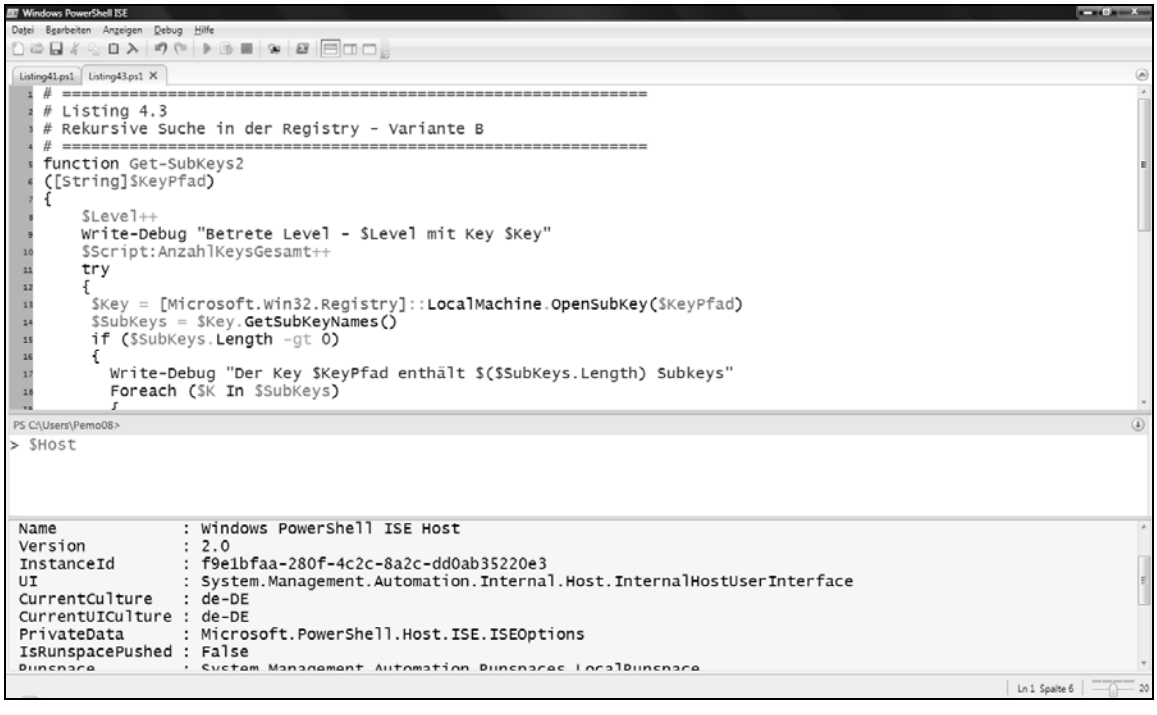

Abbildung 2.10 Der International Scripting Editor (ISE) von Microsoft – klein, funktional noch nicht ganz vollständig, aber vollkommen ausreichend für die meisten Gelegenheiten

[7] Es soll bekanntlich sogar richtige Notepad-Fans geben.

[8] Der persönliche Favorit des Autors ist nach wie vor der PowerGUI Script Editor, wenngleich die ISE durchaus ihren Reiz besitzt und fast alle Skripts dieses Buchs mit der ISE umgesetzt wurden.

TIPP Mit dem »Wischersymbol« in der Symbolleiste wird der Ausgabebereich gelöscht.

TIPP Die Suche erlaubt auch die Verwendung regulärer Ausdrücke (wenngleich eingeschränkt, da sich die verschiedenen Regex-Optionen nicht setzen lassen).

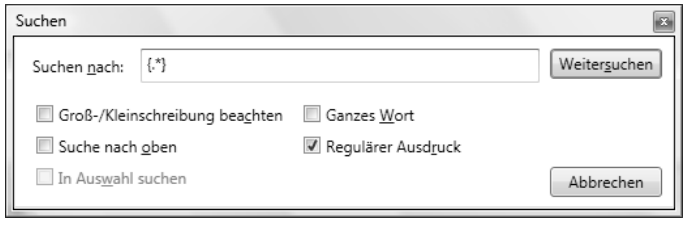

Abbildung 2.11 Bei der Suche in der ISE sind auch reguläre Ausdrücke erlaubt

HINWEIS Bei Windows Server 2008 R2 muss die PowerShell ISE als Feature nachinstalliert werden.

Die PowerShell ISE ist erweiterbar

Zur Ehrenrettung der Microsoft-Entwickler muss allerding angemerkt werden, dass die ISE auf einem Objektmodell basiert und damit (z. B. im Rahmen des Profile-Skripts) sehr flexibel und nahezu beliebig erweiterbar ist. Auf diese Weise lassen sich z. B. die Menüs erweitern (z. B. um eine Auswahlliste der zuletzt geladenen Dateien anbieten zu können) oder es können Templates eingefügt werden, sodass man nicht stets dasselbe Grundgerüst für ein Skript eintippen muss. Im Internet findet man inzwischen eine Fülle von Erweiterungen.[9] Offiziell wird das Objektmodell unter *http://technet.microsoft.com/en-us/library/dd819500.aspx* beschrieben. Ein kleines Beispiel für eine ISE-Erweiterung finden Sie in Kapitel 13.

Insgesamt steckt in der ISE eine Menge Potenzial, es muss allerdings vom Anwender erschlossen werden.

Zusammenfassung

Die PowerShell präsentiert sich als eine moderne Shell ohne Schnörkel und Ösen, deren Potenzial auch von weniger versierten Anwendern schnell ausgeschöpft werden kann. Auch wenn nichts eingestellt oder konfiguriert werden muss, bietet es sich an, eine Profile-Datei anzulegen, die bei jedem PowerShell-Start ausgeführt wird und in der z. B. Funktionen definiert oder die Farbeinstellung geändert werden. Mit Version 2.0 gibt es die PowerShell in zwei »Geschmacksrichtungen«: In der eher nüchternen Eingabeaufforderung und in der deutlich komfortableren ISE, die vor allem für das Erstellen und Ausführen von PowerShell-Skripts gedacht ist. Unter Vista und Windows 7 sollten beide Varianten bei aktivierter Benutzerkontensteuerung grundsätzlich als Administrator gestartet werden.

[9] Dabei muss man leider berücksichtigen, dass die PowerShell-Entwickler das Objektmodell nach (!) der Freigabe der CTP3 noch einmal an einigen Stellen geändert haben, sodass viele der Erweiterungen erst nach einigen Namensanpassungen funktionieren.