Monad Shell – Task-Oriented Automation Framework

Jeffrey P. Snover
Management Architect
Windows Enterprise Management Division
Jsnover @ microsoft.com

Task-Based Administrative Experience

- Tasks are the actions users perform from a
  - GUI console
  - Command line
- Example tasks
  - Add user, add disk, remove user, …
- Tasks can be comprised of sub-tasks (e.g., add user)
  - Create account in Active Directory
  - Add account to appropriate Groups
  - Create a home directory
  - …
- Administrative Experience is determined by how tasks are defined, organized, and exposed to end users

Microsoft Shell (MSH) Mission

- Deliver an extensible scripting environment that is secure, interactive, programmable, and production-ready to enable consistent and reliable automation of administrative tasks
  - Improve the developer experience by making it easier to add command-line management capabilities using .NET
  - Improve the administrative experience by enabling IT Pros to write secure automation scripts that can run locally or remotely
- Deliverables
  - A scripting language
  - An interactive shell
  - A way to produce task-oriented commands
  - A set of domain-independent utility commands
  - A mechanism to do remote scripting

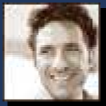
MSH Problem Statement

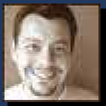
- Windows administration has not met the needs of administrators
  - Overemphasis on GUI-based tools and developer-oriented SDKs
  - Weak command shell with incomplete coverage and limited automation
- Unix employs a powerful model for automating administration tasks
  - Composition (A | B | C)
  - Text-based pipelines
    - Command A output processed by command B…
  - Uniform remoting of commands
- .NET enables Windows to do better than Unix
  - Object-based pipelines
  - Managed code
    - Commands are classes
    - Reflection-based utilities

MSH – Key Admin Scenarios

- Better than Unix Shell
  - .NET-based experience
- Compatibility and Interoperability
  - Existing commands and scripts (.exe, .bat, .vbs, …) work
- Secure Remote Scripting
  - Signed cmdlets (tiny commands) and scripts
- Configuration Settings Management
  - Get and set configuration values for desktop (network, print, Internet Explorer, …)
  - Server role deployment and operations
- Batching
  - Execute admin tasks on 1:many computers
- Seamless navigation
  - File system, Registry, AD, WMI

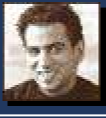
Enterprise Systems Administrator – Ray Clark


Enterprise Security Administrator – Kevin Parrish

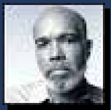

Enterprise Network Administrator – Carlos Garcia

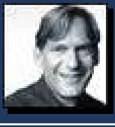
User Account Manager – Chad Rice

Windows Server Administrator – Al Young

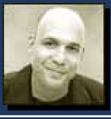

Print Administrator – Lyle Kramer

Enterprise IT

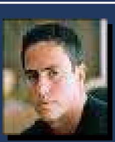

Server Systems Administrator - Sam Watson


Network Systems Administrator – Chuck Thomas

Upper MORG IT


Core MORG Operations Engineer – Chris Green

Core MORG IT

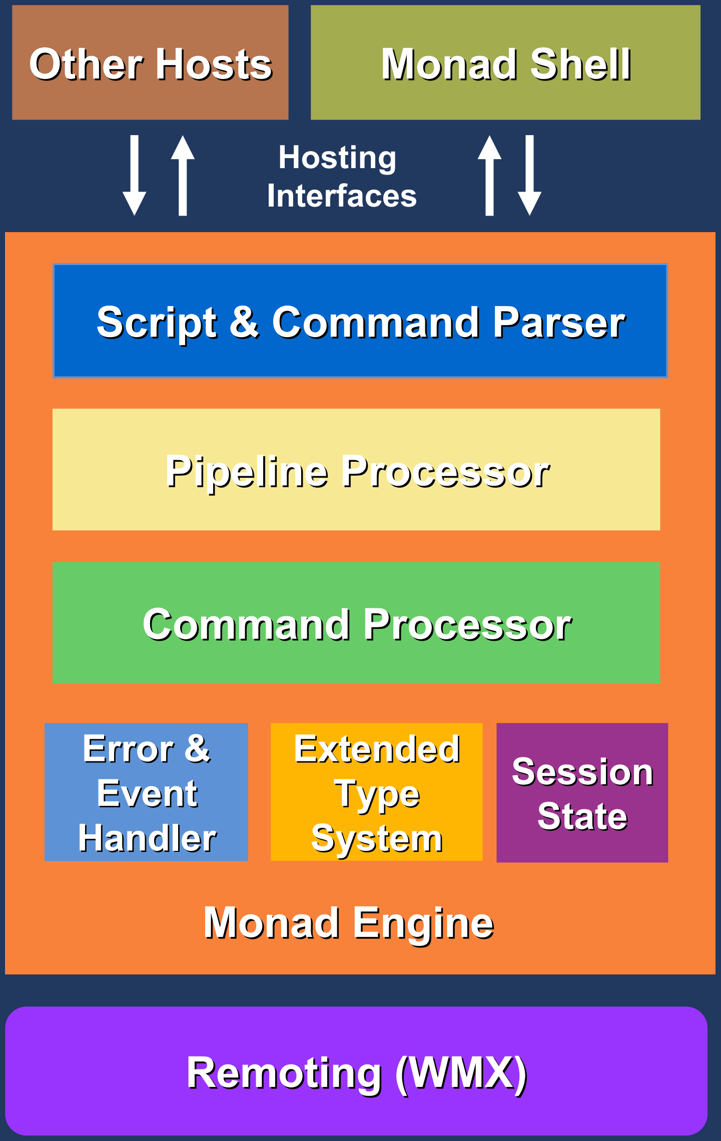

Do It Yourselfer – Frank Martinez

SORG IT

MSH Demo

- Let’s get MSH in focus
  - As interactive and composable as KSH or BASH
  - As programmable as PERL or RUBY
  - As production-oriented as VMS DCL or AS400 CL
  - Makes accessing mgmt information as easy as accessing a file system

MSH Architecture



- Monad shell (msh.exe)
  - Character-based command-line host for the Monad engine
- Monad engine (msh.dll)
  - Script/Parser – processes language constructs such as scripts, predicates, conditionals, etc.
  - Pipeline Processor – manages inter-cmdlet communication via pipes
  - Command Processor – manages cmdlet execution, registration and associated metadata
  - Session State – manages the data set used by a cmdlet for execution
  - Extended Type System – provides a common interface for accessing properties, methods, etc. independent of the underlying object type
  - Error and Event Handler – manages exception to error mapping and reporting

Key MSH Concepts For The Developer

- Cmdlets are .NET classes
  - Think DLLs not EXEs
- Providers enable groups or families of related cmdlets (i.e., namespaces)
  - File System, Registry, Active Directory, …
- Pipelines are composed of classes (cmdlets) passing structured objects
  - Objects are processed into records
- Extended Type System (ETS) simplifies developer experience
  - Common interfaces for operating on pipeline objects independent of type

Cmdlet Class

- Cmdlet class properties and methods allow cmdlets to
  - Access parameters
  - Write objects to output streams
  - Write errors
  - Access session state
  - …
- CmdletDeclarationAttribute metadata enables MSH to identify .NET class as a cmdlet
  - Requires two parameters: VerbName, NounName

```
using System.Management.Automation;
[CmdletDeclarationAttribute("get",
“process")]
class GetProcess : Cmdlet
{
implementation
}
```

Writing A cmdlet

- Cmdlet class defines three virtual methods
  - StartProcessing()
  - ProcessRecord()
  - EndProcessing()
- Cmdlets override one or more of these methods to do work
  - StartProcessing()
    - Where one-time cmdlet startup operations are performed
  - ProcessRecord()
    - Where cmdlets perform the bulk of their work
    - Processes a single object (e.g., record) at a time
  - EndProcessing()
    - Where one-time cmdlet close operations are performed

Example: Get-Process cmdlet

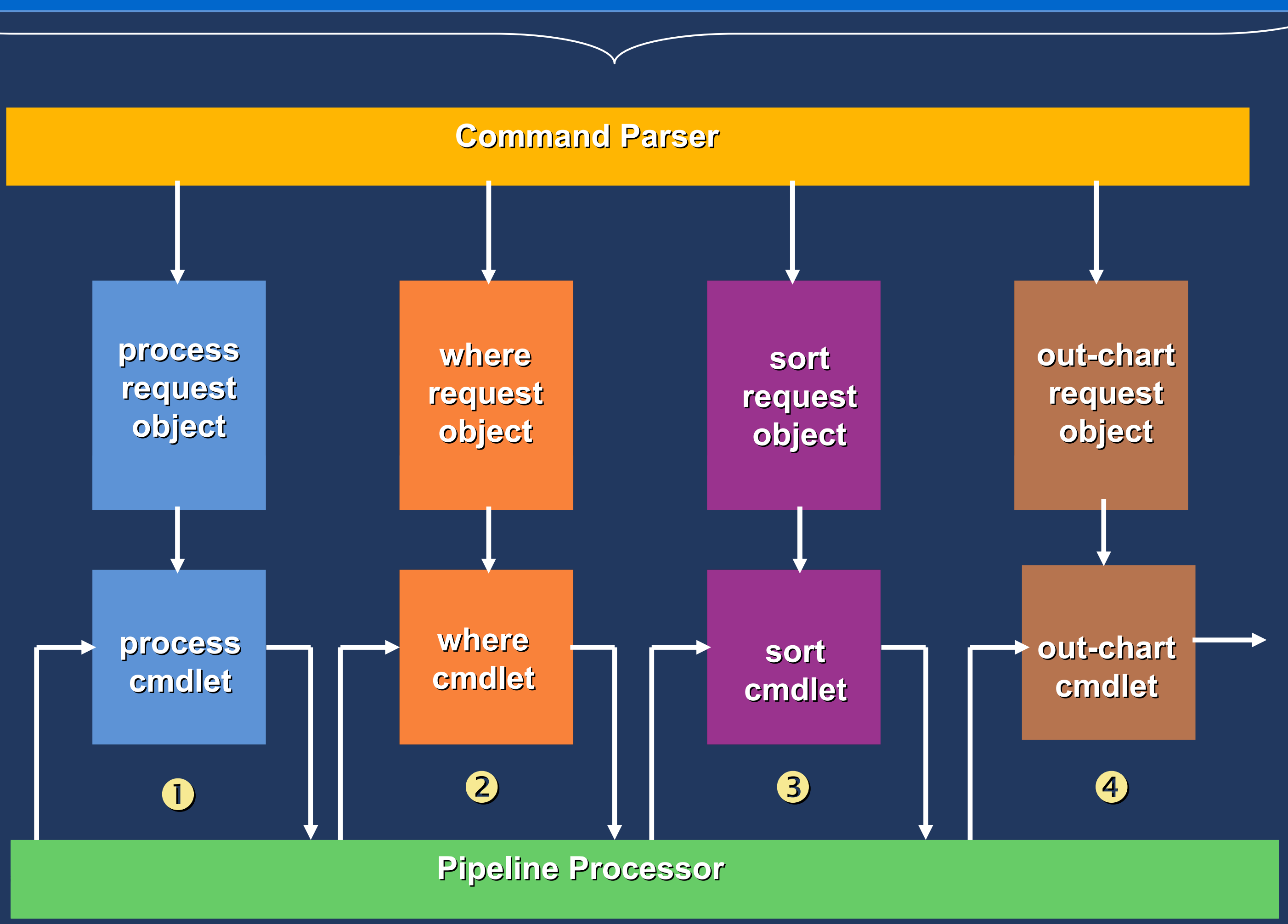
```
…
using System.Management.Automation;
[CmdletDeclarationAttribute (“get”, “process”)]
public class GetProcess: Cmdlet
{
        public override void StartProcessing()
        {
            WriteObjects (Process.GetProcess());
        }
}
```

Pipelines

- Cmdlets execute in pipelines (→A → B → C→)
  - Cmdlet attribution defines parameters for driving the parser
  - Pipeline Processor manages cmdlet execution and communication
- Cmdlets communicate indirectly through objects
  - Each cmdlet execution has its own input/output
- Cmdlets execute in same thread as pipeline
  - Remoted cmdlet executes in a separate pipeline
    - Different computer, different process
    - Input/output for remoted cmdlet is serialized between pipelines
- Cmdlets use extended reflection to operate on objects independent of type
  - MSHObject provides developers a common interface to access methods, properties, brokered methods, brokered properties, property sets, …

Pipeline Processing

```
➊ ➋ ➌ ➍
get-process | where “handlecount –gt 400” | sort handlecount | out-chart processname,handlecount
```



Parameters

- Cmdlets request parameters from
  - Command line
  - Incoming pipeline objects
- Cmdlets define parameters as fields and mark them with metadata
  - [ParsingParameterDeclaration]
  - [ParsingMandatoryParameter]
  - [ParsingAllowPipelineInput]
  - [ParsingParameterMapping(index)]
  - …
- MSH ensures parameters are filled in and validated before cmdlet ProcessRecord() method is called

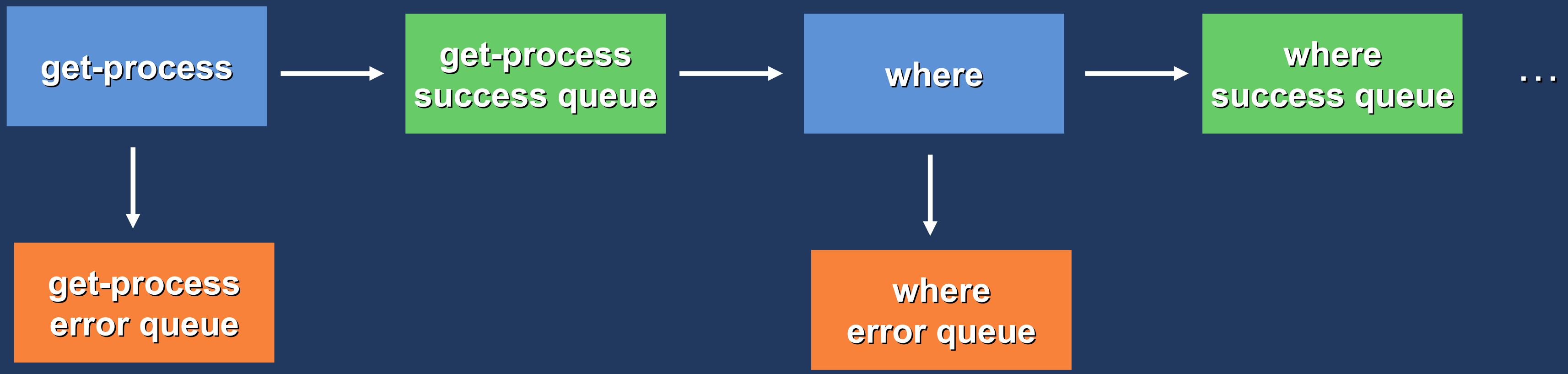
Example: Stop-Process cmdlet With Parameter

```
using System.Management.Automation
[CmdletDeclarationAttribute (“stop”, “process”)]
public class StopProcess: Cmdlet
{        [ParsingMandatoryParameter]
         [ParsingParameterMapping(0)]
         [ParsingAllowPipelineInput]
         [ParsingPromptString(“Name of the process: ")]
         public string ProcessName;

         public override void StartProcessing()
         {    Process [ ]ps;
         ps = Process.GetProcessesByName(ProcessName);
         foreach (Process p in ps)
         {    if (ShouldProcess(p.ProcessName))
                             {            p.Kill();
                             }
          }
         }
}
```

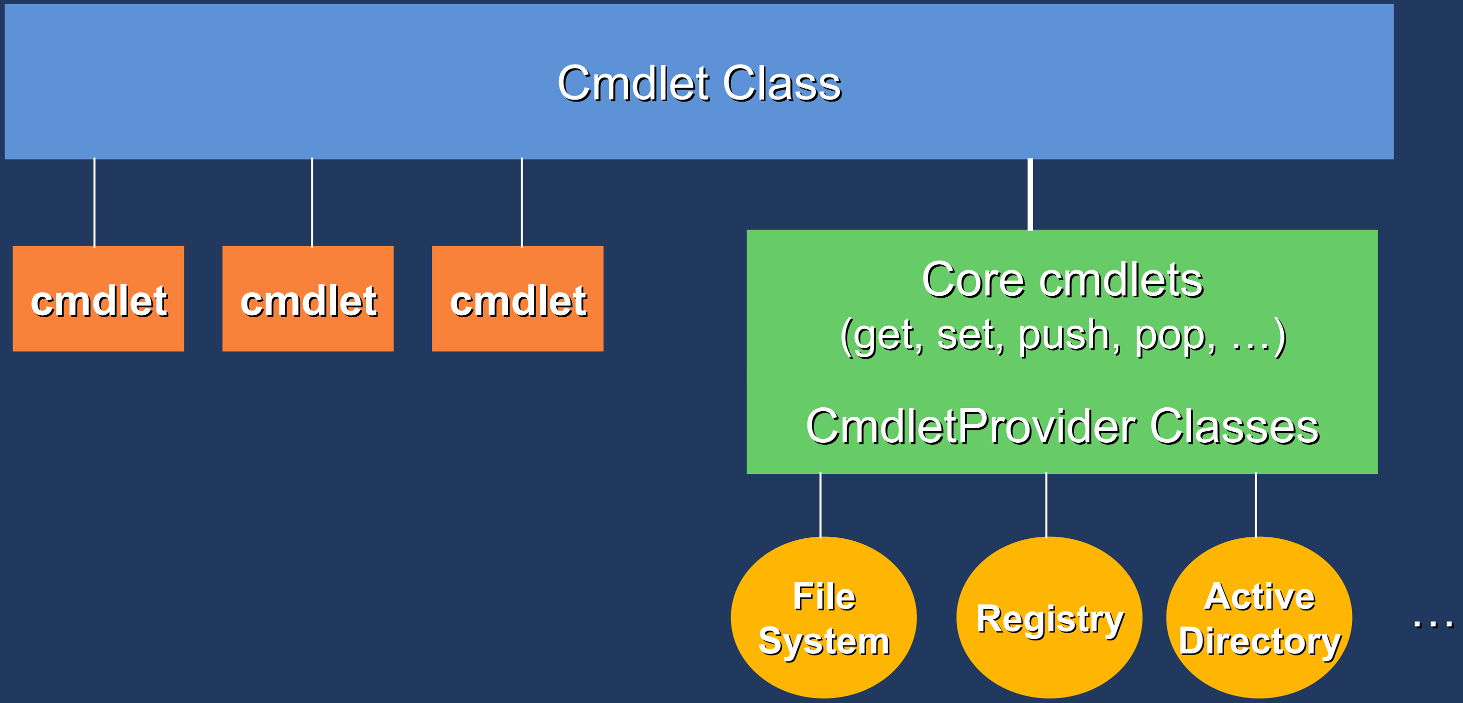
Error Handling

```
get-process | where “handlecount –gt 400” | sort handlecount | out-chart processname,handlecount
```



- Cmdlets communicate success and failure via queue objects
  - 1 input queue, 2 output queues (success, error)
  - Additional streams for verbose, progress, and debug
- Errors are first class citizens
  - Errors can be reported immediately
  - Cmdlets and pipelines can partially succeed

Cmdlet Providers



- Cmdlet class provides common interfaces for writing cmdlets
- CmdletProvider classes expose APIs for writing cmdlet providers
- Each cmdlet provider inherits a common set of core cmdlets
- Cmdlet providers should be written for
  - Configuration stores that can be navigated
  - Containers where new, move, copy, rename, and remove operations can be performed

Cmdlet/Provider Configuration And Registration

- Cmdlet file naming is verb-noun.cmdlet and contains
  - Assembly binding information
  - Help file binding information
  - Syntax (metadata) information
- Cmdlet files can be generated using export-cmdlet utility
  - Reflects on .NET assemblies to produce .cmdlet files
- Cmdlets are discovered by searching for .msh or .cmdlet files based on environment path variable settings
  - $MSHCOMMANDPATH, $PATH, $PATHEXT
- At startup MSH reads profile.msh
  - profile.msh is used to create a set of valid functions and aliases

Demo: Retrieving A List Of Running Processes

- get-process | where “handlecount –gt 400” | sort handlecount

```
ProcessName             Id   HandleCount   WorkingSet
-----------             --   -----------   ----------
csrss                  636           433      1191936
explorer              1600           447      9428992
CcmExec               1880           523     16171008
lsass                  716           543       851968
winlogon               660           644      5951488
OUTLOOK               1320          1138     38465536
svchost               1020          1401     26091520
```

- Explanation of what the above script does
  - get-process retrieves a list of running processes
  - where filters the get-process results to retain only processes with more than 400 open handles
  - sort handlecount orders the sort results by # of open handles

Demo: Using MSH To Generate A Report

- get-process | where “handlecount –gt 400” | sort handlecount | out-chart processname,handlecount

- Explanation of what the above script does
  - get-process retrieves a list of running processes
  - where filters the get-process results to retain only processes with more than 400 open handles
  - sort handlecount orders the sort results by # of open handles
  - out-chart writes the where results to an Excel chart using processname and associated handlecount values

Call To Action

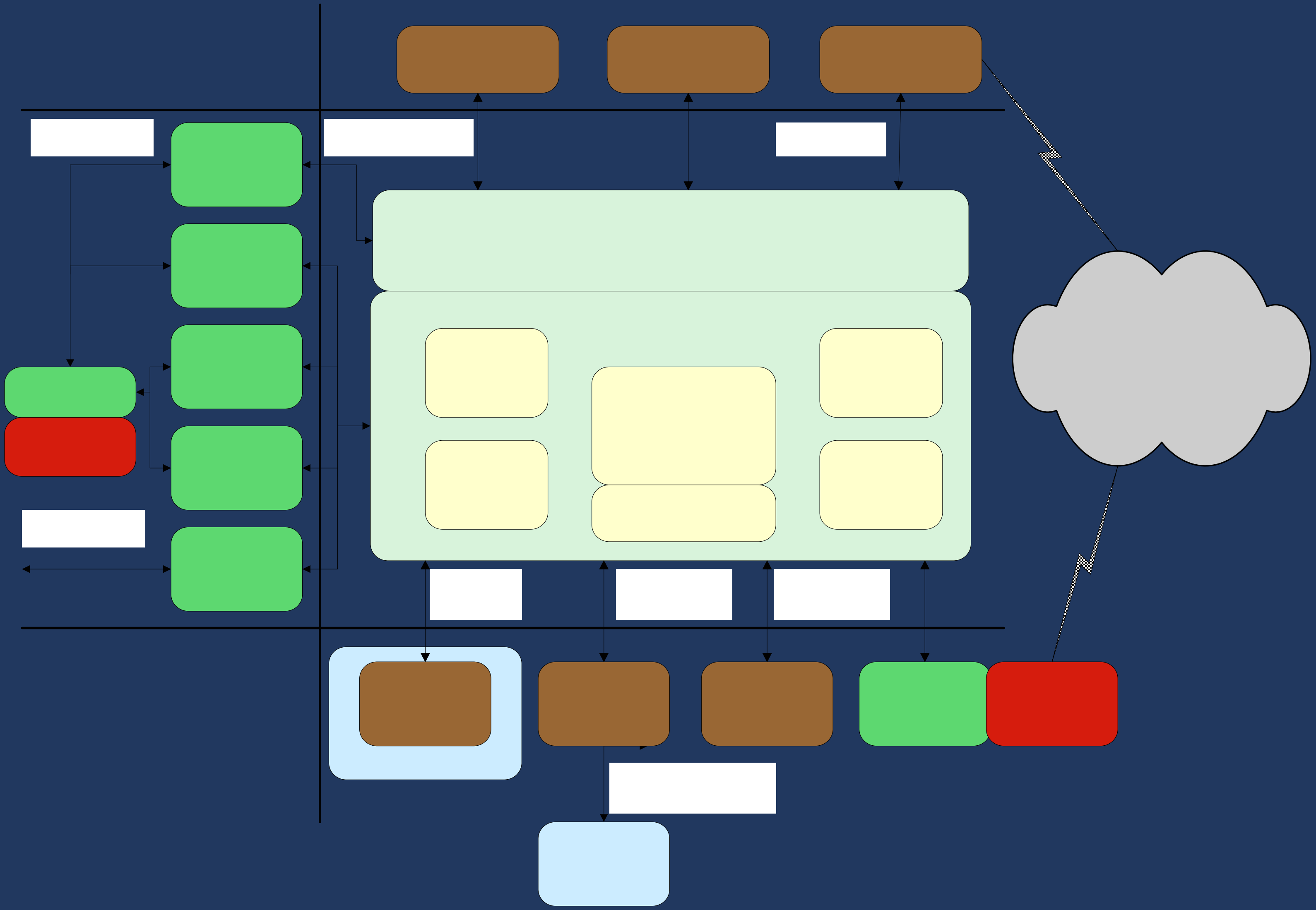
- Sign up for Command Shell Preview from betaplace
- Install it
- Use it
  - Write SCRIPTS
  - Write Cmdlets
  - Write Providers
- Give us feedback, early and often
- Help us ship the V1 that meets your needs

Additional Resources

- Web Resources
  - Available on http://betaplace.com
    Use the guest account: mshPDC
  - Logon and password e-mailed within 24 hours
  - Download bits, SDK, samples, private newsgroup, and a feedback/bug reporting environment

Question & Answer

MSH Architecture



Host Speific 24 Ho

Scripting Language

- Cmdlet syntax: <verb>-<noun> [-<qualifier> <value> [,<value>…] …]
  - Verb refers to the action
  - Noun refers to the system object
  - Qualifier-value pair refers to the parameter
- Language constructs
  - arithmetic binary operators (+, -, * /, %)
  - assignment operators (=, +=, -=, *=, /=, %=)
  - comparison operators (-eq, ==, -ne, !=, -gt, -ge, -lt, -le)
  - logical operators (!, -and, -or)
  - unary operators (++, --, +, -)
  - redirection operators (>, >>)
  - arrays and associative arrays (hash tables)
  - boolean, string
  - break, continue, return
  - comparisons
  - for, foreach, while
  - if, elseif, else
  - functions, method calls, invoke (&)
  - properties
  - variables
  - scoping

Base Cmdlets

- Providers
  - new-provider
  - get-provider
  - remove-provider
- Drives
  - new-drive
  - get-drive
  - remove-drive
- Location
  - get-location
  - set-location
  - push-location
  - pop-location
- Children
  - get-children
- Item
  - new-item
  - get-item
  - set-item
  - remove-item
  - rename-item
  - copy-item
  - move-item
  - clear-item
  - invoke-item
- Property
  - new-property
  - get-property
  - set-property
  - remove-property
  - rename-property
  - copy-property
  - move-property
  - clear-property
- Property Value
  - get-propertyvalue
  - set-propertyvalue
  - add-propertyvalue
  - remove-propertyvalue
  - clear-propertyvalue
- Content
  - add-content
  - get-content
  - set-content
  - clear-content
- Path
  - test-path
  - convert-path
  - parse-path
  - resolve-path
  - combine-path

More Cmdlets

- Process
  - get-process
  - set-process
  - stop-process
- Service
  - get-service
  - set-service
  - start-service
  - stop-service
- Pipeline
  - pick-object
  - sort-object
  - group-object
  - measure-object
  - compare-object
- Environment
  - get-environment
  - set-environment
- Help
  - get-help
- Alias
  - new-alias
  - get-alias
  - set-alias
  - remove-alias
- History
  - get-history
  - eval-history
  - import-history
- Variable
  - new-variable
  - get-variable
  - set-variable
  - add-variable
  - remove-variable
- File
  - in-file
  - out-file
- Format
  - format-table
  - format-list
  - format-wide
  - format-default
  - format-object
- XML
  - convert-xml
  - test-xml
  - converto-mshxml
  - convertfro-mshxml
  - invoke-xslt
- Output
  - out-console
  - out-printer
  - out-chart
- Expressions
  - reduce-expression
  - apply-expression

And Even More Cmdlets …

- Runspace
  - new-runspace
  - wait-runspace
  - remove-runspace
  - push-runspace
  - pop-runspace
  - test-runspace
  - import-runspace
  - export-runspace
- Security
  - get-securitydescriptor
  - set-securitydescriptor
  - get-securitycontext
  - get-credential
  - set-credential
  - get-signature
  - set-signature
  - test-signature
- Console
  - get-console
  - set-console
  - write-console
  - read-console
- Utility
  - get-date
  - get-localizedstring
  - write-object
  - write-errorobject
  - set-debug
  - write-debug
  - write-verbose
  - write-progress
  - add-note
  - start-subshell
  - get-culture
  - set-culture
- Command
  - get-command
  - eval-command
  - export-command
- Configuration
  - import-assembly
  - import-typexml
  - export-typexml
  - test-typexml
  - update-typexml
  - import-displayxml
  - export-displayxml
  - test-displayxml
  - update-displayxml

Interactive-Composable

- Command-line-oriented
- Interactive experience (aliases, navigation, IntelliSense, command line editing)
- History (statement, status, and results)
- Help (rich schema and searching)
- Pipelines (.NET and structures)
- Utilities (reflection)

Demo

```
get-process
# Globbing applies to objects
get-service A*

# Descriptive names for cmds & params
start-service -ServiceName Alerter

# only need to disambiguate
stop-service -S Alerter

# You can run any existing executable
ipconfig

# You can invoke files
demo.txt

#Rich aliasing reduces typing
alias ps get-process
ps

# Rich Navigation capabilities
cd c:¥
pushd doc*¥js*¥msh*
popd
$CdPath
cd mshf*
```

```
get-history

# Object pipeline and utilities
gps |member
gps |where "handlecount -ge 400" |sort handlecount

gps |sort MainModule.FileVersioninfo.companyName,handlecount
|table -groupby MainModule.FileVersionInfo.CompanyName
processname,handlecount

gps msh |pick ProcessName -expand modules |table
processname,filename

gps |pick processname -expand modules |where "filename -like
*ntdll.dll" |table processname

gps |pick processname -expand modules |group filename |sort
count -desc |head 15 |table count:6,name:70

# we don't limit ourselves to the console window
gps |out-grid processname,id,handlecount
gps |sort handlecount |tail 10 |out-chart processname,handlecount
gps |out-excel processname,handlecount,id,workingset
```

Programmable

- Rich, typed variables (read-only, constraints, descriptions)
- Rich operators
- Control structures (C# like with access to cmds and utilities)
- Functions (positional-named-typed-constrained params)
- Object property-method access
- Hosting
- Glide path (MMC => MSH => C#)
- Efficient cmdlet development model

Demo

```
# Typed variables
$a = "string"
$a = 1,2,3,4
$a = $(get-date)
$a = {get-date }
$a.Invoke()

# Rich set of operators
$i = 2
$s = "hello"
$i * 3
$s * 3
$i += 1
$s += "world"

$i = 10
$i % 3
$s = get-date
"Today's data is {0:MM-YY-dd}" % s
```

```
#   C# like control structures
for ($i=0; $i -le 100 ; $i +=10 ) {$i }
# But still have access to cmds
foreach ($p in get-process |where "handlecount -ge 500" |sort
handlecount ) { "{0,-15} has {1,6} Handles" %
$p.ProcessName,$p.Handlecount }


# We have scripts
edit test.msh
get-console -prompt "Enter to get a list of processes"
get-process

# We have functions
edit test.msh
function t1 {
get-console -prompt "Enter to get a list of processes"
get-process
}

# Object property & method access
$s=$(new-stopwatch)
$s
$s.Start()
$s.Stop()
```

Easy To Use

- File systems are easy to use
  - Navigation and manipulation are universal
- Other stores are hard
  - Require domain-specific utilities and concepts
- How do we make other stores easy?
  - Interact with them as with file systems

Demo

```
get-drive -scope global
pushd hklm:¥software¥microsoft
dir
cd wbem
new-item  -path .¥cimom        -Name TEST1 -content "first TEST STRING" -type String
new-item  -path .¥xml¥Decoders -Name TEST2 -content "Second TEST STRING" -type String
new-item  -path .¥wmic         -Name TEST3 -content "Third TEST STRING" -type String
new-item  -path .              -Name TEST4 -content "Forth TEST STRING" -type String

get-children -recurse -include TEST*
get-children -recurse -include TEST* |remove-item

dir c:¥do*¥*¥*.msh
dir c:¥do*¥*¥*.msh -exclude *profile*

dir alias:c*
dir env:
dir variables:
dir variables:*err*
Dir AD:
```

Production Oriented

- Uniform syntax, formatting, outputting, and processing
- Strong style guide
  - Naming
  - Errors
  - Targeting
- Admin friendly (Whatif, Confirm, Verbose)
- Rich error support ($error, -errvar, -errorpolicy, error pipelines)
- Remote Management (Secure, 1:many)

Demo

```
gps c*,s* -exc *t,*d
gps c*,s* -exc *t,*d |stop-process -whatif
gps c*,s* -exc *t,*d |stop-process -confirm
```

```
stop-service a*
$error
stop-service a* -errvar myvar
$myvar
stop-service a* -errorpolicy notifycontinue
stop-service a* -errorpolicy silentcontinue
stop-service a* -errorpolicy notifystop
stop-service a* -errorpolicy inquire
```