VMware vCenter Orchestrator Plug-In Development Best Practices

vCenter Orchestrator 4.2

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-000686-00

vmware®

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

Contents

VMware vCenter Orchestrator Plug-In Development Best Practices

VMware vCenter Orchestrator Plug-In Development Best Practices provides information about the common structure of a vCenter Orchestrator plug-in as well as helpful techniques and practices for plug-in development.

Intended Audience

This information is intended for developers who want to understand the structure and content of vCenter Orchestrator plug-ins as well as understand how to avoid specific problems and improve certain aspects of the plug-ins that they develop.
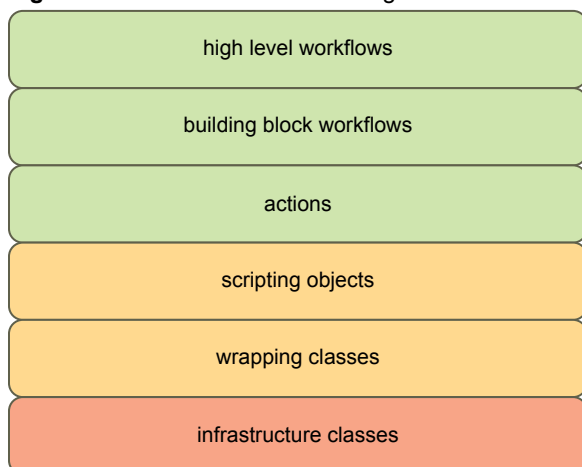
Structure of an Orchestrator Plug-In

<div style="text-align: right;">1</div>

Orchestrator plug-ins have a common structure that consists of various types of layers that implement specific functionality.

The bottom three layers of a vCO plug-in, that are, infrastructure classes, wrapping classes, and scripting objects, implement the connection between the plugged-in technology and Orchestrator.

The user-visible parts of a vCO plug-in are the top three layers that are actions, building blocks, and high level workflows.

Figure 1-1. Structure of a vCO Plug-In

Infrastructure classes	A set of classes that provide the connection between the plugged-in technology and Orchestrator. The infrastructure classes include the classes to implement according to the plug-in definition, such as plug-in factory, plug-in adaptor, and so on. The infrastructure classes also include the classes that provide functionality for common tasks and objects such as helpers, caching, inventory, and so on.
Wrapping classes	A set of classes that adapt the object model of the plugged-in technology to the object model that you want to expose inside Orchestrator.
Scripting objects	JavaScript object types that provide access to the wrapping classes, methods, and attributes in the plugged-in technology. In the `vso.xml` file you define which wrapping classes, attributes, and methods from the plugged-in technology will be exposed to Orchestrator.

Actions	A set of JavaScript functions that you can use directly in workflows, Web views, and scripting tasks. Actions can take multiple input parameters and have a single return value.
Building block workflows	A set of workflows that cover all generic functionality that you want to provide with the plug-in. Typically, a building block workflow represents an operation in the user interface of the orchestrated technology. The building block workflows can be used directly or can be included inside high-level workflows.
High level workflows	A set of workflows that cover specific functionality of the plug-in. You can provide high-level workflows to meet concrete requirements or to show complex examples of the plug-in usage.

Approaches for Building Orchestrator Plug-Ins

<div style="text-align: right; font-size: 2em;">2</div>

You can use different approaches to build your Orchestrator plug-ins. You can start building a plug-in layer by layer or you can start building all layers of the plug-in at the same time.

This chapter includes the following topics:

Bottom-Up Plug-In Development

A plug-in can be built layer by layer using bottom-up development approach.

Bottom-up development approach builds the plug-in layer by layer starting from the lower level layers and continuing with the higher level layers. When this approach is mixed with an interactive and iterative development approach, then part or whole layer is delivered for each iteration. At the end of the N iterations the plug-in is completely finished.`

Figure 2-1. Bottom-up plug-in development
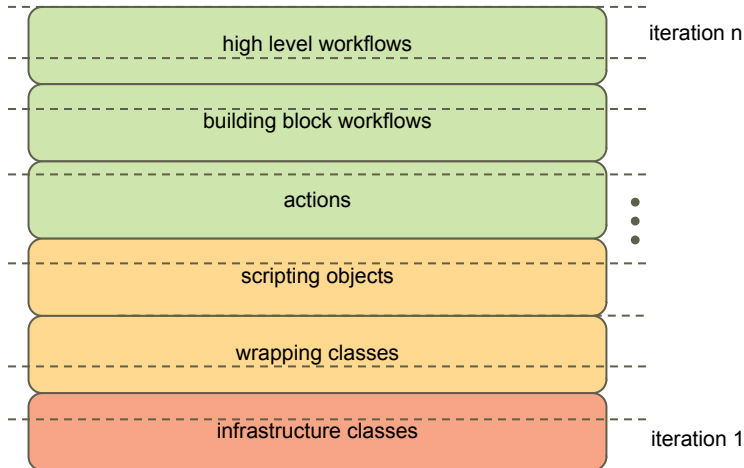
An advantage of the bottom-up plug-in development approach is that development is focused on one layer at a time.

Consider the following disadvantages of bottom-up plug-in development approach.

- The progress of the plug-in development is difficult to show until some insertions are completed.

■   It does not fit very well in an Agile development practices.

The bottom-up development process is considered good enough for small plug-ins, with reduced or non-existent set of wrapping classes, scripting objects, actions, or workflows.

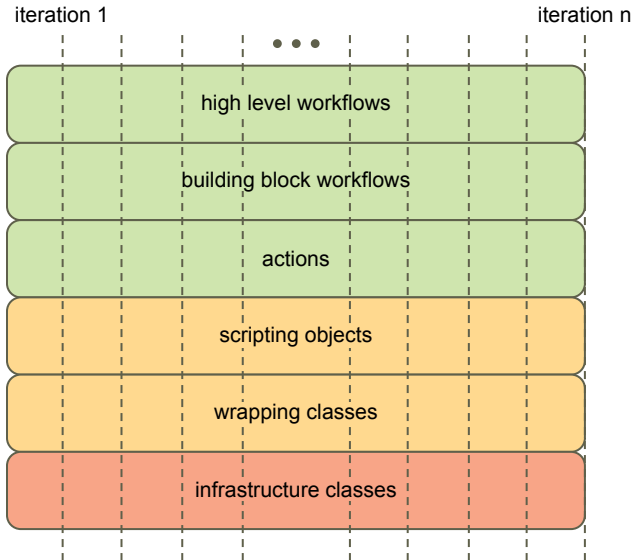# Top-Down Plug-In Development

A plug-in can be built by slicing it into top-down functionality, using top-down development approach.

When the top-down approach is mixed with an Agile development process, new functionality is delivered for each iteration. As a result, at the end of the iteration N the plug-in is completely implemented.

Figure 2-2. Top-down plug-in development

The top-down plug-in development approach has the following advantages.

■   The progress of the plug-in development is easy to show from the first iteration because new functionality is completed for each iteration and the plug-in can be released and used after every iteration.

■   Completing a vertical slice of functionality allows for very clearly defined success criteria and definition of what has been done, as well as better communication between developers, product management, and quality assurance (QA) engineers.

■   Allows the QA engineers to start testing and automating from the beginning of the development process. Such an approach results in valuable feedback and decreases the overall project delivery time frame.

A disadvantage of the top-down plug-in development approach is that the development is in progress on different layers at the same time.

You should apply the top-down plug-in development process for most plug-ins. It is appropriate for plug-ins with dynamic requirements.

Types of Orchestrator Plug-Ins 3

Using plug-ins, you can integrate in Orchestrator general-purpose libraries or utilities like XML or SSH as well as entire systems such as vCloud. Depending on the technology that you integrate in Orchestrator, plug-ins can be categorized as plug-ins for services, or general purpose plug-ins, and plug-ins for systems.

This chapter includes the following topics:

- "Plug-Ins for Services," on page 11
- "Plug-Ins for Systems," on page 12

Plug-Ins for Services

Plug-ins for services or general-purpose plug-ins provide functionality that can be considered as a service inside vCenter Orchestrator.

Figure 3-1. Archnitectire of plug-ins for services
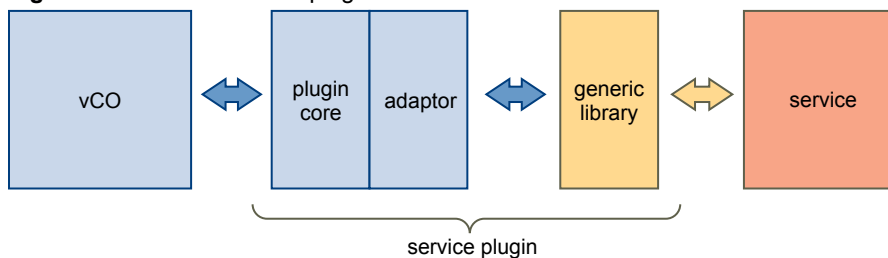
Plug-ins for services expose generic libraries or utilities to Orchestrator, such as XML, SSH or SOAP. For example, the following plug-ins that are available in Orchestrator are plug-ins for services.

JDBC plug-in	Allows you to use any database within a workflow.
Mail plug-in	Allows you to send emails within a workflow.
SSH plug-in	Allows you to open SSH connections and run commands within a workflow.
XML plug-in	Allows you to manage XML documents within a workflow.

Plug-ins for services have the following characteristics.

Complexity	Plug-ins for services have from low to medium level of complexity. Plug-ins for services expose a specific library, or part of a library, inside Orchestrator so as to provide concrete functionality. For example, the XML plug-in adds an implementation of a Document Object Model (DOM) XML parser to the Orchestrator JavaScript API.
Size	Plug-ins for services are relatively small in size. They require a basic set of classes that are the same as for all plug-ins, and other classes that offer new scripting objects to add new functionality.
Inventory	Plug-ins for services require a small inventory of objects to work, or they do not require an inventory at all. Plug-ins for services have a generic and small object model, and so, they do not need to show this model inside the vCO inventory.

Plug-Ins for Systems

Plug-ins for systems connect the vCO workflow engine to an external system so that you can orchestrate the external system.

Following are examples for plug-ins for systems.

vCenter Server plug-in	Allows you to manage vCenter Server instances using workflows.
vCloud Director plug-in	Allows you to interact with a vCloud Director installation within a workflow.
Cisco UCSM plug-in	Allows you to interact with Cisco entities within a workflow.

Following are the main characteristics of plug-ins for systems.

Complexity	Plug-ins for systems have higher level of complexity, because the technologies that they expose are relatively complex. Plug-ins for systems must represent all the elements of the external system inside Orchestrator to interact with the external system and offer the same functionality as that system in Orchestrator. If the external system provides an integration mechanism, you can use it to expose the functionality of the system in Orchestrator more easily. However, besides representing the elements of the external system in Orchestrator, plug-ins for systems might also need to offer high scalability, provide a caching mechanism, deal with events and notifications, and so on.
Size	Plug-ins for system are from medium to big in size. Plug-ins for systems require many classes apart from the basic set of classes because usually they offer a large number of scripting objects. Plug-ins for systems might require some other helper and auxiliary classes that will interact with them.
Inventory	Usually, plug-ins for systems have a large number of objects, and you must expose these objects properly in the inventory so that you can locate them and work with them easily in vCO. Because of the large number of objects that plug-ins for systems need to expose, you should build auxiliary tool or a process to auto-generate as much code as possible for the plug-in. For example, vCenter Server plug-in provides such a tool.

- Plug-Ins for Object-Oriented Systems on page 13

  Object-oriented systems offer an interaction mechanism that is based on objects and RPC.

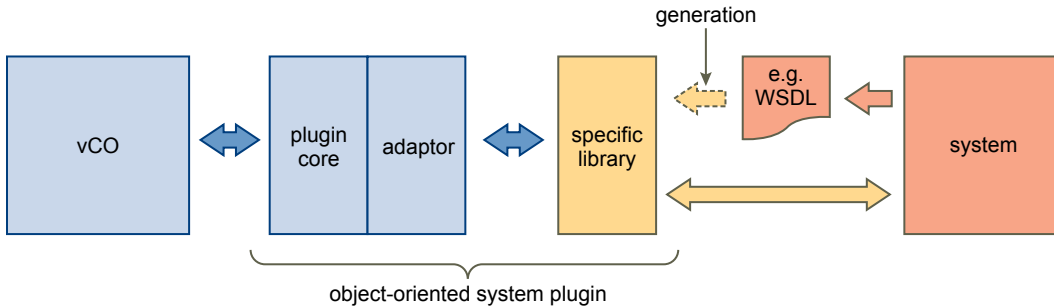■ Plug-Ins for Resource-Oriented Systems on page 13

Resource-oriented systems provide an interaction mechanism that is based on resources and simple operations that use HTTP methods.

Plug-Ins for Object-Oriented Systems

Object-oriented systems offer an interaction mechanism that is based on objects and RPC.

The most widely used model for an object-oriented system is the Web service model that uses SOAP. The objects inside this model have a set of attributes that are related to the state of the objects and offer a set of remote methods that are invoked on the target system side.

Figure 3-2. Plug-Ins for Object-Oriented Systems

You can consider the following when you implement plug-ins for object-oriented systems.

■ If you use SOAP, you can use the WSDL file to generate a set of classes that combine the object model and the communication mechanism.

■ This object model is almost everything that you have to expose inside vCenter Orchestrator.

Plug-Ins for Resource-Oriented Systems

Resource-oriented systems provide an interaction mechanism that is based on resources and simple operations that use HTTP methods.

The most representative model for a resource-oriented system is the REST model, combined for example with XML. The objects inside this model have a set of attributes that are related to their state. To invoke methods on the target system (communication mechanism), you must use the standard HTTP methods such as GET, POST, PUT, and so on, and follow some conventions.

Figure 3-3. Plug-ins for resource-oriented systems

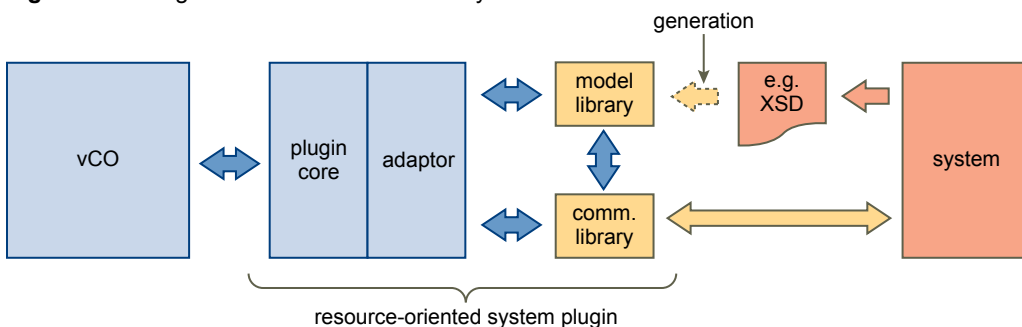You can consider the following when you develop plug-ins for resource-oriented systems.

- If you use REST or only HTTP with XML, you get one or more XML schema files to be able to read and write messages. From these schemas, you can generate a set of classes that define the object model. This set of classes only defines the state of the objects because the operations are defined implicitly with the HTTP methods, for example, as defined in the vCloud Director plug-in, or explicitly with some specific XML messages, like the Cisco UCSM plug-in.

- You need to implement the communication mechanism in another set of classes. This set of classes defines a new object model to interact with the original object model. The object model for the communication mechanism consists of objects and methods only.

- You can expose both the original object model and the object model for the communication mechanism inside vCO. This could add some extra complexity depending on how both object models are exposed, and on whether you are merging related objects from both sides (to simulate an object-oriented system) or keeping them separate.

Plug-In Implementation 4

You can use certain helpful practices and techniques when you structure your plug-ins, implement the required Java classes and JavaScript objects, develop the plug-in workflows and actions as well as provide the workflow presentation.

- **Project Structure** on page 15

  You can apply a standard structure for the projects of your Orchestrator plug-ins.

- **Project Internals** on page 16

  You can apply certain approaches when implementing your plug-in, for example, cash objects, bring object in background, clone object, and so on. By applying such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

- **Workflow Internals** on page 17

  You can implement a workflow to monitor long-time operations that your vCO plug-in performs.

- **Workflows and Actions** on page 18

  To ease the workflow development and usage, you can use certain good practices.

- **Workflow Presentation** on page 18

  When you create the presentation of a workflow, you should apply certain structure and rules.

Project Structure

You can apply a standard structure for the projects of your Orchestrator plug-ins.

You can a use a standard Maven structure with modules for your plug-in projects to bring clarity in where every peace of functionality resides.

Table 4-1. Structure of a plug-in project

Module	Description
`/myAwesomePlugin-plugin`	Root of the plug-in project.
`/o11nplugin-myAwesomePlugin`	Module that composes the final plug-in DAR file.
`/o11nplugin-myAwesomePlugin-config`	Module that contains the plug-in configuration Web application. It generates a standard WAR file.
`/o11nplugin-myAwesomePlugin-core`	Module that contains all the classes that implement any of the standard Orchestrator plug-in interfaces and other auxiliary classes that they use. It generates a standard JAR file.

Table 4-1. Structure of a plug-in project (Continued)

Module	Description
`/o11nplugin-myAwesomePlugin-model`	Module that contains all the classes that help you to integrate the third-party technology with Orchestrator through the plug-in. The classes should not contain any direct reference to the standard Orchestrator plug-in APIs.
`/o11nplugin-myAwesomePlugin-package`	Module that imports an external Orchestrator package file with actions and workflows to include it inside the final plug-in DAR file. The module is optional.

Project Internals

You can apply certain approaches when implementing your plug-in, for example, cash objects, bring object in background, clone object, and so on. By applying such approaches, you can improve the performance of your plug-ins, avoid concurrency problems, and improve the responsiveness of the Orchestrator client.

Cache Objects if Possible

Your plug-in can interact with a remote service, and this interaction is provided by local objects that represent remote objects on the service side. To achieve good performance of the plug-in as well as good responsiveness of the Orchestrator UI, you can cash the local objects instead of getting them every time from the remote service. You can consider the scope of the cache, for example, one cache for all the plug-in clients, one cache per user of the plug-in, and one cache per user of the third-party service. When implemented, your caching mechanism will be integrated with the plug-in interface for finding and invalidating objects.

Bring Objects in Background

If you have to show large lists of objects in the plug-in inventory and you do not have a fast way to retrieve those objects, you can bring objects in background. You can bring object in background, for example, by having objects with two states, *fake* and *loaded*. Assume that the *fake* objects are very easy to create and they provide the minimal information that you have to show in the inventory, like name and ID. Then it would be possible to always return *fake* objects, and when all the information (the real object) is really needed, the using entity or the plug-in can invoke a method *load* automatically to get the real object. You can even configure the process of loading objects to start automatically after the fake objects are returned, to anticipate the actions of the using entity.

Clone Objects to Avoid Concurrency Problems

If you use a cache for your plug-in, you have to clone objects. If you have a cache and you always return the same instance of an object to every entity that requests it, then unwanted effects might occur. For example, entity A requests object O and the entity views the object in the inventory with all its attributes. At the same time, entity B requests object O as well, and entity A runs a workflow that starts changing the attributes of object O. At the end of its run, the workflow invokes the object's `update` method to update the object on the server side. If entity A and entity B get the same instance of object O, entity A views in the inventory all the changes that entity B performs, even before the changes are committed on the server side. If the run goes fine, it should not be a problem, but if the run fails, the attributes of object O for entity A will not be reverted. In such case, if the cache (the `find` operations of the plug-in) returns a clone of the object instead of the same instance all the time, each using entity views and modifies its own copy, avoiding concurrency issues, at least within Orchestrator.

Notify Changes to Others

Problems might occur when you use a cache and you clone objects simultaneously. The biggest one is that the object that a using entity views might not be the last version that is available for the object. For example, if an entity displays the inventory, the objects are loaded once, but at the same time, if another entity is changing some of the objects, the first entity does not view the changes. To avoid this problem, you can use the `PluginWatcher`, `IPluginPublisher` methods from the Orchestrator plug-in API to notify that something has changed to allow other instances of Orchestrator clients to see the changes. This also applies to a unique instance of the Orchestrator client when changes from one object from the inventory affect other objects of the inventory and they need to be notified too. The operations that are prone to use notifications are adding, updating, and deleting objects when these objects, or some properties of these objects, are shown in the inventory.

Enable Finding Any Object at Any Time

You must implement the `find` method of the `IPluginFactory` interface to find objects just by type and ID. The `find` method could be invoked directly after restarting Orchestrator and resuming a workflow.

Simulate a Query Service if You Do Not Have One

The Orchestrator client can require querying for some objects in specific cases or showing them not as a tree but as a list or as a table, for example. This means that your plug-in must be able to query for some set of objects at any moment. If the third-party technology offers a query service, you need to adapt and use this service. Otherwise, you should be able to simulate a query service, despite of the higher complexity or the lower performance of the solution.

Find Methods Should Not Return Runtime Exceptions

The methods from the `IPluginFactory` interface that implement the searches inside the plug-in, should not throw controlled or uncontrolled runtime exceptions. This could be the cause of strange *validation error* failures when a workflow is running. For example, between two nodes of a workflow, the `find` method will be invoked if an output from the first node is an input of the second node. At that moment, if the object is not found because any runtime exception, probably you will get no more information than a *validation error* in the vCenter Orchestrator client. After that, it depends on how the plug-in logs the exceptions in to get more or less information inside the log files.

Workflow Internals

You can implement a workflow to monitor long-time operations that your vCO plug-in performs.

You should implement a workflow for monitoring long-time running operations such as task monitoring. This workflow can be based on Orchestrator triggers and waiting events. You have to consider that a workflow that is blocked waiting for a task can be resumed as soon as the Orchestrator server starts, the plug-in must be able to get all the required information to resume the monitoring process properly.

The monitoring workflow or the task that it can? use internally should provide a mechanism to specify the pooling rate and a possible timeout.

The process of debugging a piece of scripting code inside a workflow is not easy, especially if the code does not invoke any Java code. Because of this, sometimes the only option is to use the logging methods offered by the default Orchestrator scripting objects.

Workflows and Actions

To ease the workflow development and usage, you can use certain good practices.

Start Developing Workflows as Building Blocks

A building block can be a simple workflow that requires a few input parameters and returns a simple output. If you have a rich set of building blocks, you can create higher-level workflows easily, and you can offer a better set of tools for composing own complex workflows.

Create Higher-Level Workflows Based on Smaller Components

If you have to develop a complex workflow with lots of inputs and internal steps, you can split it in smaller and simpler building block workflows and actions.

Create Actions Whenever Possible

You can create actions to achieve additional flexibility when you develop workflows.

- To create complex objects or parameters for scripting methods easily.

- To avoid repeating common pieces of code all the time.

- To perform UI validations.

Workflows Should Invoke Actions Whenever Possible

Actions can be invoked directly as nodes inside the workflow schema. This can keep the workflow schema simpler, because you do not need to add scripting code blocks to invoke a single action.

Fill In the Expected Information

Provide information for every element of a workflow or an action.

- Provide a description of the workflow or action.

- Provide a description of the input parameters.

- Provide a description of the outputs.

- Provide a description of the attributes for the workflows.

Keep the Version Information Updated

When you version plug-ins, add meaningful comments with information such as major updates of the plug-in, important implementation details, and so on.

Workflow Presentation

When you create the presentation of a workflow, you should apply certain structure and rules.

Use the following properties for the workflow inputs in the workflow presentation.

Table 4-2. Properties for Workflow Inputs

Properties	Usage
Show in Inventory	Use this property to help the user to run a workflow from the inventory view by clicking with the right button on it.
Specify a root object to be shown in the chooser	Use this property to help the user to choose inputs easily. If the root object can be refreshed in the presentation, or it is an attribute of an attribute of, or it is retrieved by an object method, you need to create or set an appropriate action to refresh the object in the presentation.
Maximum string length	Use this property for long strings such as names, descriptions, file paths, and so on.
Minimum string length	Use this property to avoid empty strings from the testing tools.
Custom validation	Non-simple validations must be implemented with actions.

Organize the inputs with steps and display group. Such organization helps the user to identify and distinguish all the input parameters of a workflow.

Recommendations for Orchestrator Plug-In Development 5

You can consider certain recommendations when developing the different components of your Orchestrator plug-ins.

Table 5-1. Useful practices in plug-in implementation

Component	Item	Description
General	Access to third-party API	Plug-ins should provide simplified methods for accessing the third-party API wherever possible.
	Interface	Plug-ins should provide a coherent and standard interface for users, even when the API does not.
Action	Scripting objects	You should create actions for every creation, modification, or deletion of a scripting object, and for every other method that is available on the object.
	Description	The description of an action should describe what the action does instead of how it works.
	Scripting	When you use scripting to get the properties or methods of an object, you can check whether the object value is different from `null` or `undefined`.
	Deprecation	If an action is deprecated, the `comment` or the `throw` statement should indicate the replacement action, or the action should call a new replacement action so that solutions that are built on the deprecated version of the action do not break.
Workflow	User interface operations in the orchestrated technology	You should create a workflow for every operation that is available in the user interface of the orchestrated technology.
	Description	The description of a workflow should describes what the workflow does instead of how it works.
	Presentation property `mandatory input`	You need to set the `mandatory input` property for all mandatory workflow inputs.
	Presentation property `default value`	If you develop a workflow that configures an entity, the workflow presentation should load the default configuration values for this entity. For example, if you develop a workflow that is named Host Configuration, the presentation of the workflow must load the default values of the host configuration.
	Presentation property `Show in inventory`	You need to set the `Show in inventory` property to have contextual workflows on inventory objects.
	Presentation property `specify a root parameter`	You should use this property in workflows when it is not necessary to browse the inventory from the tree root.
	Workflow validation	You must validate workflows and fix all errors.

Table 5-1. Useful practices in plug-in implementation (Continued)

Component	Item	Description
	Object creation	All workflows that create a new object should return the new object as an output parameter.
	Deprecation	If a workflow is deprecated, the `comment` or the `throw` statement should indicate the replacement workflow, or the deprecated workflow should call a new replacement workflow to avoid breaking solutions that are built on previous versions of the workflow.
Inventory	Host disconnection	If your inventory contains a connection to a host and this host becomes unavailable, you should indicate that the host is disconnected. You can do this either by renaming the root object by appending - *disconnected* or by removing the tree of objects underneath this object, in the same manner as the vCloud Director plug-in does.
	`Select value as list` property	An inventory object must be selectable as `treeview` or a `list`.
	Host manager	If the plug-in implements a `host` object for the target system, then a parent `hostmanager` root object should exist with properties for adding, removing, or editing host properties.
	Getting or updating objects	If a query service is running on the orchestrated technology, you should use it for mass getting objects.
	Child discovery	If you need to retrieve object children separately, the retrieval process must be multithreaded and nonblocking on a single error.
	vCenter Orchestrator object change	All workflows that can change the state of an element in the inventory must update the inventory to avoid having objects out of synchronization.
	External object change	You can use a notification mechanism to notify about changes in the orchestrated technology that occur as a result of operations that are performed outside of vCO. In case such operations lead to removal of objects from the orchestrated technology, you must refresh the inventory accordingly to avoid failures or loss of data. For example, if a virtual machine is deleted from vCenter Server, the vCenter Server plug-in updates the inventory to remove the object of the removed virtual machine.
	Finder object	Finder objects should have properties that can be used to differentiate between objects. These are typically the properties that are present in the user interface.
Scripting object	Implementation	The `equals` method must be implemented to insure that `==` operation works on the same object as in some cases the object may have two instances.
	Plug-in object properties	Objects that have parent objects should implement a `parent` property.
	Plug-in object properties	Objects that have child objects should implement `get` methods that return arrays of child objects.
	Inventory objects	Inventory objects should be searchable with `Server.find`.
		All inventory objects should be serializable so they can be used as input or output attributes in a workflow.
	Constructor and methods	In most cases, scriptable objects should have either a constructor, or should be returned by other object attributes or methods.

Table 5-1. Useful practices in plug-in implementation (Continued)

Component	Item	Description
	Object ID	Objects that have an ID that is issued from an external system, should use an internal ID to ensure that no ID collision occurs when you are orchestrating more than one server.
	Searching for objects	`search` or `find` methods should implement a filter so that the specified name or ID can be found instead of just all objects. For example, the vCO server has a `Server.FindForId` method that allows finding a plug-in object by its ID. To do this, the method must be implemented for each findable object in the plug-in.
	Trigger	If possible, triggers should be available for objects that change so that vCO may have policies triggered on various events. For example, a trigger or an event in the vCenter plug-in on the `Datacenter` object could be monitored by vCenter Orchestrator to determine when a new virtual machine is added, powered on, powered off, and so on.
	Object properties	Objects that reside in other plug-ins should have properties for being easily converted from one plug-in object to another. For example, virtual machine objects need to have a `moref` (managed object reference ID).
	Session manager	If you are connecting to a remote server that has a possibility for different session, the plug-in should implement a shared session and a session per user.
Trigger	Trigger	All long operations and blocking methods should be able to start asynchronously with a task returned, and generate a trigger event on completion.
Enumerations	Enums	Enumerations for a given type should have an inventory object that allows selecting from the different values in the enumeration.
Logging	Logs	Methods should implement different log levels.
Versioning	Plug-in version	Plug-in version should follow the standards and be updated along with the plug-in update.
API documentation	Methods	Methods that are described in the API documentation should never throw the exception `no xyz method / property` on an object. Instead, methods should return `null` when no properties are available and be documented with details when these properties are not available.
	vso.xml	All objects, methods, and properties must be documented in `vso.xml`.

Documenting Plug-In User Interface Strings and APIs

6

When you write user interface (UI) strings for vCO plug-ins and the related API documentation, it is best practice to follow the accepted rules of style and format.

General Recommendations

- Use the official names for any of VMware products involved in the plug-in. For example, use the official names for the following products and VMware terminology.

Correct Term	Do Not Use
vCenter Server	VC or vCenter
vCloud Director	vCloud
virtual machine	VM

- End all complete sentences with a period. For example, `Creates a new Organization.` is a workflow description.

- Use a text editor with a spell checker to write the descriptions and then move them to the plug-in.

- Ensure that the name of the plug-in exactly matches the approved third-party product name that it is associated with.

Workflows and Actions

- Write informative descriptions. One or two sentences are enough for most of the actions and workflows.

- Higher-level workflows might include more extensive descriptions and comments.

- Start descriptions with a verb, for example, `Creates…`. Do not use self-referential language like `This workflow creates…`.

- Put a period at the end of descriptions that are complete sentences.

- Describe what a workflow or action does instead of how it is implemented.

- Workflows and actions usually are included in folders and packages. The enclosing folders and packages should include a small description as well. For example, a workflow folder can have a description similar to `Set of workflows related to vApp Template management`.

Parameters of Workflows and Actions

- Start workflow and action descriptions with a descriptive noun phrase, for example, `Name of…`. Do not use a phrase like `It's the name of…`.

- Do not put a period at the end of parameter and action descriptions. They are not complete sentences.

- Input parameters of workflows must specify a label with appropriate names in the presentation view. In many cases, you can combine related inputs in a display group. For example, instead of having two inputs with the labels Name of the Organization and Full name of the Organization, you can create a display group with the label Organization and place the inputs Name and Full name in the Organization group.

- For steps and display groups, add descriptions or comments that appear in the workflow presentation as well.

Plug-in API

- The documentation of the API refers to all of the documentation in the `vso.xml` file and the Java source files.

- For the `vso.xml` file, use the same rules for the descriptions of finder objects and scripting objects with their methods that you use for workflows and actions. Descriptions of object attributes and method parameters use the same rules as the workflow and action parameters.

- Avoid special characters in the `vso.xml` file by including the descriptions inside a `<![CDATA[insert your description here!]]>` tag.

- Use the standard Javadoc style for the Java source files.

Index