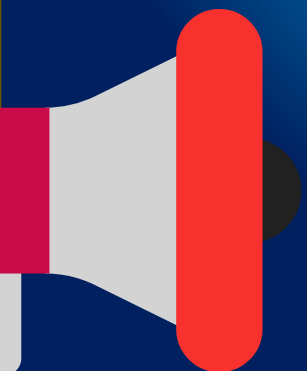


TIPS & TRICKS TO MASTER THE SOLIDWORKS



PDM API



- › 50 TIPS & TRICKS
- › PDM API CODE EXAMPLES IN C#, VB.NET
- › SAVE HOURS OF RESEARCH & DEBUGGING
- › BECOME A BETTER API DEVELOPER.

BLUE BYTE SYSTEMS INC.

Blue Byte Systems Inc., based in **Vancouver, Canada**, specializes in delivering innovative solutions for the **SOLIDWORKS** and **SOLIDWORKS PDM** ecosystems. Our expert team focuses on automating **workflows**, enhancing efficiency, and minimizing operational costs for businesses of all sizes. We provide a comprehensive range of services, including custom **API solutions**, tailored macros, PDM data migration, and online training programs. Our API classes and hands-on training empower clients to maximize the potential of SOLIDWORKS and PDM, ensuring they stay ahead in a competitive landscape.

ABOUT THE AUTHOR



- Founder and Technical Director of Blue Byte Systems Inc.
- SOLIDWORKS API User Group Leader.
- Author of multiple courses and open-source frameworks for the SOLIDWORKS API.
- Best Practices Guide for SOLIDWORKS PDM and the CSWP-API certification exam.
- Expert in delivering custom API solutions, PDM data migration, macros, and online training programs.

TIPS & TRICKS TO MASTER THE SOLIDWORKS PDM API



Overview

Welcome to “**Tips & Tricks to Master the SOLIDWORKS PDM API**” – a practical guide designed to empower you with tips, insights, and shortcuts that will enhance your development experience with the SOLIDWORKS PDM API.

Whether you are building add-ins, automating workflows, or integrating PDM with external systems, this eBook will equip you with actionable advice to streamline your projects, save time, and avoid common pitfalls.

Who is This Guide For?

This guide is tailored for developers, system integrators, and advanced administrators who have some level of familiarity with the SOLIDWORKS PDM API. To get the most out of this guide, you should:

- **Understand the Basics of the SOLIDWORKS PDM API:**
Knowledge of PDM concepts like vaults, workflows, and permissions is essential. Familiarity with writing add-ins or scripts using the PDM API will help you apply these tricks effectively.
- **Have Experience with Development Tools:**
Comfort with development environments such as Visual Studio, and a basic understanding of C# or VB.NET, will make the examples and explanations easier to follow.
- **Be Actively Working with SOLIDWORKS PDM Projects:**
Whether you are maintaining an existing system or building a new integration, this guide will provide you with insights to elevate your solutions.

What's Inside?

The tricks in this guide cover a wide range of topics, including:

- **Best Practices for Writing Add-ins:**
Learn how to structure and optimize your PDM add-ins for performance and maintainability.
- **Common Pitfalls and How to Avoid Them:**
Save time by avoiding mistakes that even experienced developers make.
- **Debugging and Troubleshooting Tips:**
Gain insights into effectively debugging your code and resolving issues with PDM API integrations.

Why This Guide?

The SOLIDWORKS PDM API is a powerful tool, but like any API, it comes with its quirks and challenges. This guide condenses years of experience into practical tricks that you can immediately apply to your work. Think of it as a reference manual for real-world solutions that go beyond what the official documentation provides.

TIP 01: What is the difference between the IEdmVault5 login methods?

In the **SOLIDWORKS PDM API**, the three main login methods are **LoginAuto**, **Login**, and **LoginEx**. Each serves a different purpose for authenticating users to the PDM vault programmatically. Here is an explanation of each:

LoginAuto

The **LoginAuto** method piggybacks on the current SOLIDWORKS PDM session. This method does not require a username or password, which is a **significant advantage**. If there is no PDM session, the blue login dialog box will be displayed, prompting the user to log in.

Use this method if:

- Your application is a **standalone** or VBA script.
- Your application is **user-attended**.
- It is part of a **PDM add-in** or a **dispatch script**. The host application already has an-going PDM session.
- Your application can achieve all its intended goals with the permissions of the currently **logged-in user** (e.g., folder and workflow access).

LoginEx

The **LoginEx** method is used when the calling application acts as an external agent logging into the vault. This method ensures that a PDM license is consumed, thereby complying with the End-User License Agreement (EULA). Unlike **LoginAuto**, this method requires a username and password.

Use this method if:

- Your application/script is **unattended**.
- You want to ensure a PDM license is consumed.
- You need access to the vault under a **different** username.

Login

Attempts a login to the specified vault name with a username and a password but does not ensure a PDM license is consumed.

TIP 02: What is Hwnd?

Hwnd appears in many PDM API calls as a parameter and is unfortunately not well-defined in the SOLIDWORKS PDM API documentation.

Hwnd

Handle of window: a unique identifier in Win32.

It identifies a window in File Explorer and is not persistent across different sessions of File Explorer (explorer.exe). This means that the window handle may change for the window.

In the context of the SOLIDWORKS PDM API, the window handle is especially important because it allows:

- The blue login dialog box is to be shown **atop** the host application window. If you specify **zero** in the **LoginAuto** method, the login dialog box may **NOT** be rendered in the right area of your screen causing the user to be confused.

The host application's control will be given away to the login box which is not visible to the user. A common consequence is the host application window flashing and not responding to the user input.

- A similar behavior may concur as well for API calls such as the check in API call. The check in API call may trigger some PDM dialog boxes to render in the wrong area of the screen if **Hwnd** is not specified.

💡 HOW TO GET THE WINDOW HANDLE IN .NET:

In WinForms, use to get the handle of the form.

In a Console app, use

`Process.GetCurrentProcess().MainWindowHandle.ToInt32()`

In a WPF app, use `new WindowInteropHelper(this).Handle.ToInt32()` to get the handle of the active window.

TIP 03: File ID and Folder ID go together

SOLIDWORKS PDM is a file-based system. All files are stored in folders and thus whenever an API construct has an ID of a file, it most likely will also have the ID of the folder where it the file resides.

Whenever you define a variable that will hold the value of a file ID, make sure to create another one to hold the ID of the folder. Many usual API calls for utilize the file ID and will require you supply the folder ID.

A good example of this is that [GetLocalPath](#) function that is part of the [IEdmFile5](#) interface. This method requires that you specify the folder ID in order to return the fully qualified path name of the file.

The same is true for the check in, check out API calls.

When the vault notifies your add-in about an event that affects a specific file, the [EdmCmdData](#) which is the structure that identifies the affected document almost always have the file ID **AND** the folder ID.

TIP 04: Always use new interfaces

The SOLIDWORKS PDM API is built with backward compatibility in mind. All interfaces start with Edm (Electronic Data Management) and end in a number. A good example of this is [IEdmVault5](#). The five refers to the version of SOLIDWORKS PDM.

The new interface version implements the obsolete version but not vice versa. [IEdmVault6](#) will have all the methods from [IEdmVault5](#) in the code autocomplete dropdown in Visual Studio.

Using the latest version of the interface will avoid doing unnecessary casts between the obsolete version and the updated version throughout your code.

TIP 05: PDM Errors

The SOLIDWORKS PDM API does not define an exception class.

All PDM errors are thrown by the API as com exceptions. This exception's message will have the error message in the current locale. When catching exceptions, it is customary to go from the most specific exception to the most generic one.

Your program might contain other CAD programs such SOLIDWORKS 3D. In this case, the SOLIDWORKS API might throw a [COMException](#) as well. To find out if the [COMException](#) is coming from PDM, use the method [GetErrorString](#) with the error code parameter being the [HRESULT](#) property from the [COMException](#).

If the returned description has a value, then the error is coming from SOLIDWORKS PDM.

TIP 06: Searching is faster than iterating

When getting all the files in a folder, you have **two options**: Traverse the content of the folder or perform a search for all *.sld* files.

To traverse the content of a folder in PDM, you can leverage the **iterator pattern** like in the example below:

```
var file = default(IEdmFile5);
var pos = folder.GetFirstFilePosition();
While (pos.IsNull == false) {
    Console.WriteLine(file.Name);
    file = folder.GetNextFile(pos);
}
```

You can get the same output by **searching** the folder below:

```
var folder = default(IEdmFolder5);
var searchEngine = vault.CreateSearch();
searchEngine.FileName = $"*.sld*";
searchEngine.FindFiles = true;
searchEngine.Recursive = false;
searchEngine.StartFolderID = folder.ID;
searchEngine.FindFolders = false;
var searchResult = searchEngine.GetFirstResult();
while (searchResult != null)
{
    if (searchResult.IsKindOf(EdmObjectType.EdmObject_File))
    {
        WriteLine(searchResult.Name);
    }
    searchResult = searchEngine.GetNextResult();
}
```

The search approach is **considerably faster** than iterating over the content. This becomes more visible when you are working with a large folder.

TIP 07: Avoid race conditions when saving files directly to the vault

A **race condition** occurs when two or more lines access shared resources simultaneously, and the outcome depends on the specific timing or sequence of their execution. Race conditions can lead to unpredictable and undesirable behavior in a program.

This scenario is a fitting example of a race condition in the context of SOLIDWORKS PDM:

Context

1. A program (for example SOLIDWORKS) saves a file to a folder in PDM. The file's extension is part of the list of files to add automatically.
2. PDM automatically detects the new file and adds it to the vault, making it accessible through its API (e.g., [GetFileFromPath](#)).
3. Shortly after saving the file, the program attempts to retrieve the file using [IEdmVault5::GetFileFromPath](#).

Problem

If the program calls [GetFileFromPath](#) before PDM has finished adding the file to the vault, the function will fail because the file is not yet recognized as part of the PDM vault.

This sequence of events depends on timing:

1. The speed at which the PDM processes and adds the file.
2. The speed at which the program proceeds to call [GetFileFromPath](#).

If PDM is slower, the logic fails, resulting in:

- An error indicating the file does not exist in the vault.
- A failure to retrieve or process the file further.

Race Condition Explained

The shared resource here is the "existence of the file in the PDM vault."

- One process (PDM) is asynchronously adding the file to the vault.
- Another process (the program) is trying to access the file through the vault API.

The race condition occurs because:

- The program assumes the file will be available immediately after saving it.
- There is **no guarantee** that PDM will process the file before the program's `GetFileFromPath` call.

Solution

To resolve this, you need to ensure the program waits for PDM to finish adding the file to the vault before calling `GetFileFromPath`.

Two approaches can help:

1. Polling Mechanism:

- After saving the file, periodically check if the file exists in the vault using `GetFileFromPath`.

Example:

```
Do
    Set file = vault.GetFileFromPath(filePath)
    If Not file Is Nothing Then Exit Do
    Sleep(100) ' Wait for 100ms before trying again
Loop
```

2. Save file to a temporary folder:

The best solution is to save the file to a temporary folder and then use `AddFile` method part of the `IEdmFolder5` interface since it returns the id of added file.

TIP 08: Writing Add-ins with Version-Specific APIs

When writing an add-in for SOLIDWORKS PDM, it is essential to consider compatibility across different versions of the software. Some API calls introduced in newer versions may not be available in older versions, potentially causing runtime errors for clients using older systems.

Scenario

Suppose your add-in uses an API feature introduced in SOLIDWORKS PDM 2022, but some clients are still using SOLIDWORKS PDM 2018. To ensure compatibility, you can programmatically check the PDM version installed on the client's machine before calling the new API.

Solution: Use the `GetVersion` Method

The `GetVersion` method allows you to retrieve the major and minor version numbers of the installed SOLIDWORKS PDM Professional. Based on the version, your code can decide whether to execute the new API call or fall back to an alternative logic for older versions.

Implementation

Here is an example in Visual Basic:

```
Sub CheckPDMVersionAndCallAPI()  
    Dim vault As IEdmVault5 = New EdmVault5()  
    Dim majorVersion As Integer = 0  
    Dim minorVersion As Integer = 0  
  
    ' Get the installed PDM version  
    vault.GetVersion(majorVersion, minorVersion)  
  
    If majorVersion >= 2022 Then
```

```

        ' Call the new API method
        CallNewAPIFeature()
    Else
        ' Fallback logic for older versions
        MsgBox("The new API is not supported on this version of
SOLIDWORKS PDM. Please upgrade to 2022 or later.")
    End If
End Sub

Sub CallNewAPIFeature()
    ' Implement the new API feature here
    MsgBox("Calling the new API feature available in SOLIDWORKS
PDM 2022.")
End Sub

```

This approach ensures your add-in remains robust and user-friendly, even in environments with mixed PDM versions.

TIP 09: Flushing Changes When Editing or Reading Card Variables

When working with card variables in SOLIDWORKS PDM, it is crucial to flush changes to ensure the values are properly saved. Without this step, any modifications made to variables may not persist.

Key Steps:

1. Getting Configuration Names:

- Use `IEdmFile5::GetConfigurations` to retrieve the list of configuration names for the file.
- If implementing this in your add-in's `IEdmAddIn5::OnCmd` method, cast the `ppoData` parameter to `EdmCmdData`. The `EdmCmdData.mpoExtra` field contains an `IEdmStrLst5` of configuration names.

2. Editing or Reading Variables:

- After making updates to the variables, ensure you call either:
 - `IEdmEnumeratorVariable5::Flush`, or
 - `IEdmEnumeratorVariable8::CloseFile` with the `bFlush` argument set to `True`.

💡 READING VARIABLES FROM DATABASE:

Use `GetVarFromDB` to get the latest value of the variable without having to flush. `GetVar` and `SetVar` will read what is **locally cached**.

TIP 10: Extracting a "Parts Only" BOM from

IEdmFile7::GetComputedBOM

When using `IEdmFile7::GetComputedBOM`, the method returns an **indented BOM**, which includes assemblies and their child components. To generate a **"Parts Only" BOM**, you must filter out assembly nodes and retain only the leaf nodes (individual parts).

TIP 11: Enforcing Minimum PDM Version Compatibility in Your Add-In

When developing add-ins for SOLIDWORKS PDM Professional, it is important to ensure compatibility with the correct PDM version. Using the [EdmAddInInfo](#) structure, you can prevent your add-in from loading in older PDM versions where certain API calls might not function properly.

How To Set Minimum Required Version:

Use the [mlRequiredVersionMajor](#) and [mlRequiredVersionMinor](#) fields of the [EdmAddInInfo](#) structure to specify the minimum major and minor versions of SOLIDWORKS PDM required for your add-in.

Example:

When developing add-ins for SOLIDWORKS PDM Professional, certain command types, such as [EdmCmdType.EdmCmd_PreBrowseFolder](#) and [EdmCmdType.EdmCmd_PostBrowseFolder](#), are only available starting from PDM 2024. To ensure your add-in does not load in older versions that lack these APIs, you must enforce version compatibility using the [EdmAddInInfo](#) structure.

TIP 12: Simplifying Updates for Custom Task Add-Ins

When deploying custom task add-ins in SOLIDWORKS PDM, streamline the update process by dedicating a single machine as the task host. This ensures efficient updates and reduces the complexity of maintaining multiple systems.

Use a Dedicated Task Machine:

- Assign one machine exclusively for running task add-ins.
- Avoid deploying task add-ins across multiple machines to simplify updates and minimize maintenance overhead.

Centralize Updates:

- When an updated version of the add-in is released, update only the dedicated task machine.
- This prevents the need to update and verify installations across multiple machines in the environment.

Minimize Downtime:

- Centralized updates allow you to quickly roll out changes or fixes without disrupting workflows on other machines.

TIP 13: Canceling Workflow Transitions with Add-Ins

In SOLIDWORKS PDM, normal add-ins can cancel a workflow transition by setting `poCmd.Cancel = 1` in `OnCmd` method. However, task add-ins do not have the capability to cancel workflow transitions directly.

Task add-ins operate differently and cannot directly cancel a workflow transition.

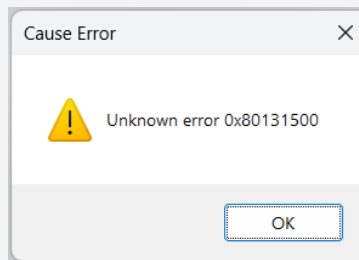
Tasks are typically executed as separate processes and are not designed to interact directly with workflow state transitions in real-time.

It is **critical** to add subscribe to the `EdmCmdType.EdmCmd_PreState` hook to be able to cancel the state change.

TIP 14: Wrapping the implementation of `OnCmd` and `GetAddInInfo` in a Try-Catch Block

When implementing SOLIDWORKS PDM add-ins, always wrap your `OnCmd` and `GetAddInInfo` methods in a try-catch block. This practice ensures that unhandled exceptions do not cause generic PDM errors, improving reliability and user experience.

`throw new Exception("This is unhandled error");` will throw a cryptic error message:



💡 BE AWARE OF ASSEMBLY RESOLUTION EXCEPTIONS:

Assembly resolution is a big topic .NET. Whenever your add-in is loaded in the PDM session, the underlying .NET framework will attempt to find the main project references. Exceptions related to assembly resolution failure will be thrown during the creation of the add-in class and will not be captured by the try catch block. Of the `OnCmd` and `GetAddInInfo` methods. Most of these exceptions will have an error message like: **The specified file could not be found.**

💻 CODE EXAMPLE:

https://github.com/BlueByteSystemsInc/PDM_API_Tips_Tricks

TIP 15: Avoid Making API Calls from Worker Threads

In multi-threaded applications, the **main thread** is the primary thread where the application starts execution and typically handles user interface (UI) interactions. A **worker thread** is a secondary thread created in a multi-threaded application to handle tasks that run concurrently with the main thread. Unlike the main thread, which typically manages user interface (UI) interactions, worker threads are used for performing background operations, such as file processing, network requests, or other computational tasks.

In the context of SOLIDWORKS PDM add-ins:

- Add-ins are executed by the **main thread** of the host application, such as ConisioAdmin.exe (for the PDM Administration Tool) or Explorer.exe (for the windows File Explorer).

The **main thread** manages UI interactions and initializes the PDM add-ins. It ensures synchronization between the user interface and underlying operations. Add-ins are invoked by the main thread of the host application, making it safe to interact with the UI during the add-in's execution.

However, when additional **worker threads** are created to perform background tasks, these threads operate independently of the main thread. API calls that interact with the UI are **not thread-safe** when called from worker threads, leading to potential crashes, deadlocks, or inconsistent behavior.

A good example of this [IEdmFile15::GetThumbnail2\(\)](#)

TIP 16: `IEdmFile5::IsLocked` needs refreshing if the lock state changes.

A **dependency property** is a special type of property used in frameworks like WPF (Windows Presentation Foundation). It extends the functionality of standard CLR (Common Language Runtime) properties by supporting features like data binding, animations, and property value inheritance.

One of the key benefits of dependency properties is their ability to **automatically update** when their underlying data changes.

PDM Properties and CLR Properties

In SOLIDWORKS PDM:

- Properties like `IEdmFile5::IsLocked` are **simple CLR properties**.
- They do not have the automatic update capabilities of dependency properties. Instead, these properties rely on cached data and require explicit actions to refresh their values when the underlying state changes.

Why `IEdmFile5::IsLocked` Needs `IEdmFile5::Refresh`

The `IsLocked` property does not automatically update when the file's state changes in the PDM vault. To ensure you get the current state, you must explicitly call `IEdmFile5::Refresh`. Without this step, you might work with outdated information.

TIP 17: Avoid Using PostLock, PreLock Hooks in PDM when possible.

When developing SOLIDWORKS PDM add-ins, avoid hooking into events like **PostLock**, **PreLock**, unless necessary. These hooks are triggered very frequently during typical vault operations, which can lead to performance issues and unnecessary complexity.

Why to Avoid These Hooks:

- **Frequent Invocation:** These events are triggered every time a user performs related actions (e.g., locking/unlocking files, adding files to the vault). For large vaults or frequent operations, this can result in excessive overhead.
- **Performance Impact:** Running custom logic for every invocation can slow down PDM operations and create a bottleneck for users.

Better Alternatives:

- Use **workflow transitions** as a trigger for custom logic instead. Workflow transitions are less frequent, easier to manage, and inherently tied to specific states and conditions in the vault.

If You Must Use These Hooks:

- Scope the logic to a specific folder to minimize impact.
- Narrow the logic down to essential files or conditions to reduce unnecessary processing.

TIP 18: Commands, Hooks and Limited to Add-Ins

In SOLIDWORKS PDM, the [IEdmCmdMgr5](#) interface provides the ability to add custom commands and hooks into the PDM environment.

This functionality is **not available for standalone applications**. Custom commands and toolbar menu items can only be added through **PDM add-ins**.

TIP 19: Running PDM Tasks from Standalone Applications with RunTask

Starting from SOLIDWORKS PDM 2018, standalone applications can run PDM tasks directly using the [RunTask](#) method of the [IEdmTaskMgr](#) interface.

This new capability allows standalone programs to initiate task add-ins, providing more flexibility for automating PDM workflows without needing a full add-in.

TIP 20: Using `EdmMenuFlags` to Customize Commands with `IEdmCmdMgr5::AddCmd`

The `EdmMenuFlags` enumeration allows you to control how and where your custom commands appear in the SOLIDWORKS PDM interface when using the `IEdmCmdMgr5::AddCmd` method. Below are tips on how to use specific combinations of these flags effectively.

Control The Command's Menu Item

To create a command that appears only in the **File menu**, requires a selection, applies to single files, and is not displayed in the context menu. Here is a list of commonly used combinations.

Use Case	Flag Combination	Integer Value
Single File	<code>EdmMenu_OnlyInContextMenu +</code> <code>EdmMenu_MustHaveSelection +</code> <code>EdmMenu_OnlyFiles +</code> <code>EdmMenu_OnlySingleSelection</code>	43
Multiple Files	<code>EdmMenu_OnlyInContextMenu +</code> <code>EdmMenu_MustHaveSelection +</code> <code>EdmMenu_OnlyFiles +</code> <code>EdmMenu_OnlyMultipleSelection</code>	51
Single Folder	<code>EdmMenu_OnlyInContextMenu +</code> <code>EdmMenu_MustHaveSelection +</code> <code>EdmMenu_OnlyFolders +</code> <code>EdmMenu_OnlySingleSelection</code>	77
Multiple Folders	<code>EdmMenu_OnlyInContextMenu +</code> <code>EdmMenu_MustHaveSelection +</code> <code>EdmMenu_OnlyFolders +</code> <code>EdmMenu_OnlyMultipleSelection</code>	85

TIP 21: Using Singleton Pattern for Efficient Searches with CreateSearch2

The [CreateSearch2](#) method in SOLIDWORKS PDM is a powerful tool for performing advanced searches with multi-variable conditions and special syntax. However, creating a new search object repeatedly can lead to unnecessary resource consumption and degraded performance. To address this, implement a **singleton pattern** to ensure a single instance of the search object is created and reused throughout your application.

```
public class SearchManager
{
    private static SearchManager _instance;
    private static readonly object _lock = new object();
    private IEdmSearch9 _searchObject;

    // Private constructor to prevent instantiation
    private SearchManager(IEdmVault5 vault)
    {
        // Instantiate the search object once
        _searchObject = (IEdmSearch9)vault.CreateSearch2();
    }
    // Public method to get the singleton instance
    public static SearchManager GetInstance(IEdmVault5 vault)
    {
        if (_instance == null)
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new SearchManager(vault);
                }
            }
        }
    }
}
```

```
        }  
    }  
    return _instance;  
}  
// Accessor for the search object  
public IEdmSearch9 GetSearchObject()  
{  
    return _searchObject;  
}  
}
```

TIP 22: Creating Submenus in SOLIDWORKS PDM with IEdmCmdMgr5::AddCmd

You can create submenus in SOLIDWORKS PDM by using the `\\` separator in the menu name when calling `IEdmCmdMgr5::AddCmd`. This allows for better organization of custom commands within the right-click context menu in Windows File Explorer.

Example:

```
poCmdMgr.AddCmd(15065, "ParentMenu\\ChildMenu", 43, string.Empty, string.Empty, -1, 0);
```

Important Notes:

1. **Context Menu Only:** Submenus created with this method will only appear in the **right-click (context) menu** of Windows File Explorer.
2. **No Modification of Existing Menus:** You cannot add submenus to existing SOLIDWORKS PDM menus or add-ins. This technique is strictly for custom commands.

TIP 22: Using `poCmd.Cancel` as a Gatekeeper to Control Workflow in SOLIDWORKS PDM

The `poCmd.Cancel` method is a powerful tool for controlling the flow of your data in SOLIDWORKS PDM add-ins. By invoking this method, you can function as a **gatekeeper**, stopping specific actions from proceeding unless certain conditions are met with all command types that start with *Pre*.

Here is an example of a PDM add-in that controls which files are allowed to transition by hooking into the `EdmCmd_PreState` event and using `poCmd.Cancel = 1` to prevent invalid files from proceeding.

```
public class TransitionControlAddIn : IEdmAddIn5
{
    public void GetAddInInfo(ref EdmAddInInfo poInfo, ref
IEcmdMgr5 poCmdMgr)
    {
        // Set add-in information
        poInfo.mbsAddInName = "Transition Control Add-In";
        poInfo.mbsCompany = "BLUE BYTE SYSTEMS INC.";
        poInfo.mbsDescription = "Controls which files can
transition based on custom rules.";
        poInfo.mlAddInVersion = 1;

        // Specify that the add-in hooks into events
        poCmdMgr.AddHook(EdmCmdType.EdmCmd_PreState);
    }

    public void OnCmd(ref EdmCmd poCmd, ref EdmCmdData[]
ppoData)
    {
        // Hook into the PreStateChange event
        if (poCmd.meCmdType == EdmCmdType.EdmCmd_PreState)
```

```

    {
        foreach (var data in ppoData)
        {
            // Get the file's path
            string filePath = data.mbsStrData1;

            // Example: Prevent files with "Draft" in
            the name from transitioning
            if (filePath.Contains("Draft"))
            {
                // Cancel the transition
                poCmd.mbCancel = 1;
                return;
            }
        }
    }
}

```

TIP 23: Using Scheduled Tasks for Predefined Folder Actions in SOLIDWORKS PDM

Scheduled tasks in SOLIDWORKS PDM are an awesome way to automate a set of predefined actions on a folder. Scheduled tasks are especially useful for tasks like:

- Regularly updating folder contents.
- Performing batch operations.

TIP 24: Use Immutable IDs to Identify Objects in SOLIDWORKS PDM

In the SOLIDWORKS PDM API, all objects, such as files, are assigned an immutable ID that remains constant.

Using these IDs to identify objects is a best practice as they do not change, unlike file paths or names.

This approach ensures that your scripts and add-ins handle files accurately and remain robust against changes in file structure or naming conventions. Always rely on IDs for consistent and error-free object identification in your PDM workflows.

TIP 26: Overwriting the Latest Version in SOLIDWORKS PDM

Using `IEdmFile5::UnlockFile`

When unlocking a file or checking it in using the `IEdmFile5::UnlockFile` method, you can utilize the `EdmUnlock_OverwriteLatestVersion` flag (value 64) to overwrite the latest version of the file in the vault without creating a new version. This is particularly useful in scenarios where you want to update the file without increasing version history.

Key Details:

1. Flag Usage:

The `EdmUnlock_OverwriteLatestVersion` flag is passed as part of the `UnlockFlags` parameter in the `UnlockFile` method.

2. Scenario:

Use this flag when minor changes need to be applied to a file, and version history updates are unnecessary (e.g., fixing metadata or small errors).

3. Caution:

- Overwriting the latest version means losing traceability for changes. Ensure this is an intentional and acceptable practice for your workflow.
- Permissions and workflow states may restrict this operation. Confirm that the user has the necessary permissions to overwrite the latest version.

4. Example Code:

```
Dim file As IEdmFile5
Dim vault As IEdmVault5

' Assuming file and vault objects are initialized and file is
checked out

Dim unlockFlags As Integer
unlockFlags = EdmUnlockFlag.EdmUnlock_OverwriteLatestVersion '
Overwrite the latest version

Try
    file.UnlockFile(hwnd, "A comment", unlockFlags, nothing)
    MsgBox("File unlocked and overwritten successfully.")
Catch ex As COMException
    MsgBox("PDM Error unlocking file: " & ex.Message)
End Try
```

TIP 27: Using `IEdmFindUser` for User Searches in SOLIDWORKS PDM

The `IEdmFindUser` interface in SOLIDWORKS PDM provides a way to perform user searches, both interactively and silently. This interface is useful for scenarios where you need to locate users within a PDM vault, such as automating user management tasks or linking card buttons to the Find User command.

Key Details:

1. Accessing the Interface:

Use the `IEdmVault7::CreateUtility` method to access `IEdmFindUser`. Set the `eType` parameter to `EdmUtility.EdmUtil_FindUser` to create an instance of the utility.

```
IEdmFindUser findUserUtility =  
(IEdmFindUser)vault.CreateUtility(EdmUtility.EdmUtil_FindUser);
```

2. Interactive Searches:

Call methods on the `IEdmFindUser` interface to display the same user search interface used in SOLIDWORKS PDM Professional. This can enhance usability when you need user input to select or locate users.

3. Silent Searches:

For automation, you can perform silent searches without displaying the user interface. This is ideal for backend operations or when integrating with custom workflows.

4. Practical Use Cases:

- Automating user-related tasks such as assigning permissions or updating metadata.
- Enhancing card functionalities by linking buttons to user searches.
- Integrating user lookup features in custom add-ins.



TIP 28: Using the Iterator Pattern in the SOLIDWORKS PDM API

The **Iterator Pattern** is a powerful design pattern that enables sequential access to elements in a collection without exposing the collection's underlying structure. In the SOLIDWORKS PDM API, this pattern can be particularly useful when traversing large datasets, such as files, folders, or search results.

Implementing the Iterator Pattern

1. Common Use Case: Iterating Through Files in a Folder

Use the [IEdmFolder5::GetFirstFilePosition](#) and [IEdmFolder5::GetNextFile](#) methods to iterate through all files in a folder.

```
Sub IterateFilesInFolder(vault As IEdmVault5, folderPath As String)
    ' Get the folder object
    Dim folder As IEdmFolder5
    Set folder = vault.GetFolderFromPath(folderPath)

    If folder Is Nothing Then
        MsgBox "Folder not found!"
        Exit Sub
    End If

    ' Initialize file position
    Dim filePos As IEdmPos5
    Set filePos = folder.GetFirstFilePosition

    ' Iterate through files
    Dim file As IEdmFile5
    While Not filePos.IsNull
        Set file = folder.GetNextFile(filePos)
        Debug.Print "File Name: " & file.Name
    End While
End Sub
```

```
Wend  
End Sub
```

2. Advanced Use Case: Iterating Through Search Results

Use the `IEdmSearch5::GetFirstResult` and `IEdmSearch5::GetNextResult` methods to iterate through search results.

```
Sub IterateSearchResults(vault As IEdmVault5, searchCriteria As  
String)  
    ' Create a search object  
    Dim search As IEdmSearch5  
    Set search = vault.CreateSearch2  
  
    ' Set search parameters  
    search.FileName = "*" & searchCriteria & "*"  
  
    ' Initialize search position  
    Dim result As IEdmSearchResult5  
    Set result = search.GetFirstResult  
  
    ' Iterate through search results  
    While Not result Is Nothing  
        Debug.Print "File: " & result.Name & " | Path: " &  
result.Path  
        Set result = search.GetNextResult  
    Wend  
End Sub
```

Key Considerations

- Always check for Null or Nothing when iterating, as empty collections may return Null positions or results.
- For large datasets, consider implementing error handling or progress reporting to ensure smooth execution.
- The iterator pattern improves maintainability by decoupling the iteration logic from the data structure.

TIP 29: Using NuGet Packages for PIA References Instead of the SOLIDWORKS PDM Installation Folder

When developing applications with the SOLIDWORKS PDM API, you typically reference Primary Interop Assemblies (PIAs) to interact with COM-based APIs. While these assemblies are available in the SOLIDWORKS PDM installation folder, it's better to use NuGet packages to embed the interop DLLs for your projects.

Benefits of Using NuGet Packages

1. **Portability:** NuGet packages embed the required interop DLLs directly into your project, ensuring the application can run independently of the specific version of SOLIDWORKS PDM installed on the development machine.
2. **Version Management:** NuGet packages allow you to specify and lock the exact version of the PDM API you are targeting, reducing compatibility issues.
3. **Simplified Setup:** Developers don't need to manually locate and reference the PIAs from the PDM installation folder, which can vary by system or user environment.

How to Use NuGet Packages for PIA References

- Open your project in Visual Studio.

- Go to Tools > NuGet Package Manager > Manage NuGet Packages for Solution....
- Search for “BlueByte” or similar packages containing the SOLIDWORKS PDM interop DLLs. Blue Byte Systems Inc. maintains a few nuget packages of the SOLIDWORKS and PDM interops. They are listed as [BlueByte.SOLIDWORKS.Interop](#) and [BlueByte.SOLIDWORKS.PDMProfessional.Interop](#).
- Install the appropriate package for your project.

TIP 30: Setting Embed Interop Types for SOLIDWORKS PDM Interop DLLs in C# and VB.NET

In C#, it's recommended to set Embed Interop Types to False for SOLIDWORKS PDM interop assemblies. This ensures that the required types are fully included in the assembly at runtime, preventing runtime errors when certain methods or interfaces are missing.

Why False?

- .NET generates interop code at runtime when Embed Interop Types is enabled (true). However, some SOLIDWORKS PDM API methods rely on extended functionality that may not be properly embedded, leading to errors.
- Setting it to False ensures all interop types are explicitly available.

How to Set It?

- In Visual Studio, right-click the interop DLL reference in the Solution Explorer.
- Select Properties.
- Set Embed Interop Types to False.

VB.NET

- In Visual Studio, you can set Embed Interop Types to true with no issues in your VB.NET project.

TIP 31: Debugging SOLIDWORKS PDM API Using the Console Debugger

The **SOLIDWORKS PDM Professional Add-In Debugger** is a powerful tool to simplify debugging your PDM API add-ins. It eliminates the need for complex setups. [You can download it from this repository on GitHub.](#)

Advantages

- Enables the use of `Console.WriteLine` for real-time debug output.
- Supports multi-selection for testing scenarios.
- No need to attach the debugger to explorer.exe or notepad.exe.

Usage

Option 1: Use Compiled Binaries

1. Download the compiled binaries from the release page.
2. Right-click the downloaded archive, select Properties, and click Unblock.
3. In Visual Studio:
 - Navigate to your project's Debug tab (C#) or Build tab (VB.NET).
 - Set the PDMAddInDebugger.exe path (located in bin/debug) as the Start with Program option.
 - *(Optional)* Add your PDM vault view local path as a command-line argument.

Option 2: Clone and Build

1. Clone the repository from the source.
2. Open the solution (.sln) file in Visual Studio.

3. Restore NuGet packages and build the project.
4. Repeat the steps from **Option 1** to configure the debugger.



TIP 32: Uncheck "Prefer 32-bit" for Standalone Applications Using SOLIDWORKS PDM API

When developing standalone applications with the SOLIDWORKS PDM API, it's crucial to ensure the **"Prefer 32-bit"** option is unchecked in your project settings. This is a recommendation made by [SOLIDWORKS](#).

Some API calls will throw a generic exception if **Prefer 32-bit** is checked. [It's an example on a cadoverflow.com](#).

How to Uncheck "Prefer 32-bit" in Visual Studio

1. Open your project in Visual Studio.
2. Navigate to Project Properties > Build.
3. Locate the "Prefer 32-bit" checkbox under the Platform Target section.
4. Ensure "Prefer 32-bit" is unchecked.
5. Save your changes and rebuild your application.

TIP 33: Understanding When the `GetAddInInfo` Method is Loaded in SOLIDWORKS PDM

The `GetAddInInfo` method is triggered at specific moments during the use of SOLIDWORKS PDM, initializing add-ins or loading additional information. This helps explain why vault login times can sometimes be longer.

Key Scenarios:

1. Adding an Add-In to a Vault:

- When an add-in is added to a vault, the `GetAddInInfo` method is invoked to register its details.

2. First Login to the Vault:

- When a user logs into the vault for the first time in a session, `GetAddInInfo` loads to prepare the add-in for use.

Performance Tip:

Avoid adding time-consuming logic to the `GetAddInInfo` method, as it runs during vault login. Keeping it lightweight ensures faster login times and a smoother user experience.

TIP 34: Server-Client Deployment of PDM Add-Ins

A SOLIDWORKS PDM add-in follows a **server-client architecture**, where the add-in is centrally managed on the server and distributed to clients as needed. This ensures consistency and ease of deployment across all users

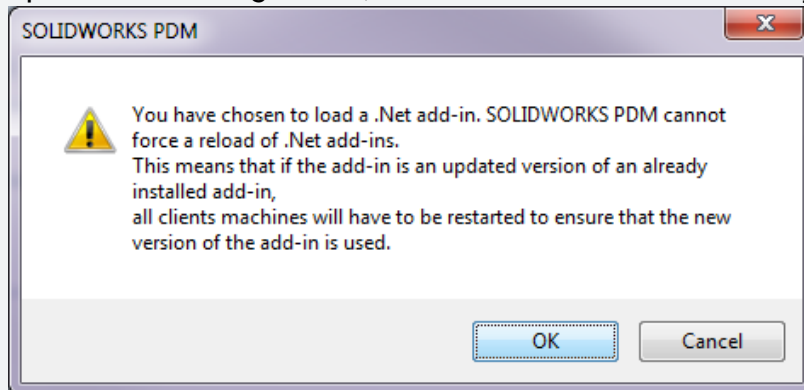
How Add-Ins Are Managed

1. Server-Side Deployment:

- Add-ins are initially added to the PDM server using the PDM Administration tool.
- The server stores all registered add-ins and controls their distribution to clients connected to the vault.

2. Client-Side Synchronization:

- When a user logs into the PDM vault, the client checks for updates or new add-ins on the server.
- If an add-in is missing or updated, the client downloads it from the server.
- PDM cannot force a restart of your client sessions so when you update an existing add-in, the administration tool warns you:



3. Local Storage:

- Add-ins are copied locally to the client machine.
- They are saved in the following directory:
`C:\Users\<username>\AppData\Local\SOLIDWORKS\SOLIDWORKS PDM\Plugins\<VaultName>\<GUID>`
- The folder name is derived from the globally unique identifier (GUID) of the add-in to ensure uniqueness and prevent conflicts.

4. Add-In Execution:

- Once copied, the add-in runs locally on the client machine as part of the PDM interface, responding to commands, events, and workflows triggered by the user.

TIP 35: Using IEdmDictionary5 for Secure and Flexible Settings Storage in SOLIDWORKS PDM

The [IEdmDictionary5](#) interface in the SOLIDWORKS PDM API provides a convenient and secure way to store and retrieve settings. These settings can be specific to your add-in or user, and they are protected by the PDM vault's login system.

Advantages

1. **Secure Storage:** Data is tied to the PDM vault and secured by the user's login credentials.
2. **Add-In-Specific Settings:** Use a unique key to isolate settings for your add-in.
3. **User-Specific Settings:** Make settings unique to each user by appending the username to the key.

Storing Settings

To store a setting in the dictionary, use the [StringSetAt](#) method:

```
// Get the vault object and dictionary
IEdmVault5 vault = /* Get your vault object */;
IEdmDictionary5 dictionary =
    vault.GetDictionary("MyAddInDictionary", true);

// Serialize your settings to a JSON string
string jsonSettings = JsonConvert.SerializeObject(new { Theme =
    "Dark", AutoSaveInterval = 10 });

// Use a key to identify the setting
```

```
string addInKey = "MyAddIn_Settings";

// Store the serialized JSON string in the dictionary
dictionary.SetStringAt(addInKey, jsonSettings);
```

Retrieving Settings

To retrieve the stored settings, use the `StringGetAt` method:

```
// Retrieve the JSON string from the dictionary
string retrievedJsonSettings;
dictionary.StringGetAt(addInKey, out retrievedJsonSettings)
// Deserialize the JSON string back into an object
var settings =
JsonConvert.DeserializeObject<dynamic>(retrievedJsonSettings);
```

User-Specific Settings

You can create user-specific settings by appending the username to the dictionary key:

```
string userSpecificKey =
$"MyAddIn_Settings_{vault.CurrentUser.Name}";
dictionary.SetStringAt(userSpecificKey, jsonSettings);
```

Best Practices

1. **Use Descriptive Keys:** Clearly define your dictionary keys to avoid conflicts.
2. **Validate Data:** Ensure your JSON strings are valid before storing them.
3. **Avoid Excessive Use:** Reserve the dictionary store for data that needs to be secure or specific to the PDM environment.
4. **Secure Storage:** Data is tied to the PDM vault and secured by the user's login credentials.

5. **Add-In-Specific Settings:** Use a unique key to isolate settings for your add-in.
6. **User-Specific Settings:** Make settings unique to each user by appending the username to the key.



TIP 36: Cancel certain events in PDM.

In SOLIDWORKS PDM, normal add-ins can cancel specific event `poCmd.Cancel = 1` in `OnCmd` method provided you subscribe to the right hooks in the `GetAddInInfo` method.

You can also cancel tasks during the `EdmCmd_TaskLaunch` hook.

Here's a concise list of **events** in SOLIDWORKS PDM API you can cancel. You can only actions/events that occur in your local SOLIDWORKS vault.

Command Type	Description
EdmCmd_PreExploreInit	Triggered before a new instance of File Explorer is created. Allows customization of vault view tabs.
EdmCmd_PreAdd	Triggered before adding a file to the vault. Useful for validating files or preparing for the addition.
EdmCmd_PreAddFolder	Triggered before adding a folder to the vault. Provides an opportunity to handle pre-creation logic.
EdmCmd_PreCopy	Triggered before copying a file. Allows custom processing or validation before the copy operation.
EdmCmd_PreCopyFolder	Triggered before copying a folder. Enables pre-copy logic or checks for folders.
EdmCmd_PreDelete	Triggered before deleting a file. Useful for validation or logging.
EdmCmd_PreDeleteFolder	Triggered before deleting a folder. Allows pre-deletion checks or actions.
EdmCmd_PreGet	Triggered before retrieving a file from the vault. Offers customization of the retrieval process.

EdmCmd_PreLabel	Triggered before creating a label. Useful for validating label creation or customization.
EdmCmd_PreLabelDelete	Triggered before deleting a label. Enables checks or additional logic before deletion.
EdmCmd_PreLabelModify	Triggered before modifying a label. Allows for validation or pre-modification processing.
EdmCmd_PreLabelAddItem	Triggered before adding an item to a label. Useful for validation or custom logic.
EdmCmd_PreLock	Triggered before checking out a file. Enables pre-check-out validation or preparation.
EdmCmd_PreMove	Triggered before moving a file. Allows pre-move validation or actions.
EdmCmd_PreMoveFolder	Triggered before moving a folder. Enables validation or preparation for folder movement.
EdmCmd_PreRename	Triggered before renaming a file. Allows checks or custom logic before renaming.
EdmCmd_PreRenameFolder	Triggered before renaming a folder. Useful for pre-rename validation.
EdmCmd_PreShare	Triggered before sharing a file from one folder to another. Allows for pre-share logic.
EdmCmd_PreState	Triggered before changing the state of a file in a workflow. Useful for enforcing rules or validation.
EdmCmd_PreUndoLock	Triggered before undoing a check-out (equivalent to check-in of an unmodified file).
EdmCmd_PreUnlock	Triggered before checking in a modified file. Enables pre-check-in processing.

TIP 37: Using Advanced Search Syntax in SOLIDWORKS PDM for Precise Queries

To enhance your SOLIDWORKS PDM Professional searches, utilize the advanced syntax available through the [IEdmVault21::CreateSearch2](#) method. This enables the use of logical operators (AND, OR, NOT), comparison operators (=, <, >, <=, >=, <>), wildcards (*, ?), and variable bindings (@VariableName) to construct precise and efficient search queries.

Key Features of the Advanced Search Syntax

1. **Logical Operators:** Combine multiple conditions using AND, OR, and NOT.
 - Example: To find files with 'Approved' status and the word 'Project' in the title: `@Status='Approved' & @Title='Project'`
2. **Comparison Operators:** Filter data using operators such as =, <, >, <=, >=, and <>.
 - Example: To search for files with a version greater than 5:
`@Version > 5`
3. **Wildcard Characters:**
 - Use * for multiple character matches.
 - Use ? for single character matches.
 - Example: Search for files starting with 'Design_' followed by any characters: `Design_*`
4. **Variable Bindings:** Directly reference variable names in your search queries using the @VariableName syntax.

- Example: Find files where the 'Document Number' contains '567' and the 'Comment' includes 'xyz': `@DocumentNumber=567 & @Comment='xyz'`
5. Date and Numeric Filters: Create precise searches by specifying date ranges or numeric values.
- Example: Files modified after January 1, 2025: `@ModifiedDate > '2025-01-01'`

TIP 38: Optimizing Large-Scale Operations with Batch Interfaces in SOLIDWORKS PDM

When making numerous calls to the same operation in SOLIDWORKS PDM, the batch interfaces (e.g., [IEdmBatchAdd2](#), [IEdmBatchChangeState5](#)) are designed to maximize efficiency and performance.

Here's a list all of the batch interfaces in PDM and their intended uses.

Interface	Description
IEdmBatchAdd2	Allows adding multiple files or folders to the vault with improved support for additional options.
IEdmBatchChangeState5	Enables batch transitioning of files to a different workflow state with the most advanced features.
IEdmBatchDelete3	Provides functionality for deleting multiple files from the vault with enhanced validation.
IEdmBatchItemGeneration2	Facilitates batch generation of items, ensuring efficient handling of linked item creation.
IEdmBatchListing4	Retrieves detailed lists of files, folders, or items in the vault with extended filtering options.
IEdmBatchUnlock2	Check-in operation for multiple files in a batch, including support for conditional operations and error handling.
IEdmBatchUpdate2	Updates properties or variables for multiple files in the vault with better performance and reliability.

TIP 39: Restarting SOLIDWORKS PDM for developers

When restarting SOLIDWORKS PDM, it's important to ensure that all related applications consuming PDM resources are completely closed. Failure to do so may prevent add-ins from updating. You can close these applications from the Task Manager's Details tab.

Applications to Close

1. **EdmServer.exe:** This is the primary process for the PDM server. Ensure it is stopped before restarting.
2. **Windows Explorer (explorer.exe):** Windows Explorer interacts heavily with PDM through the vault view. Restart it to release PDM resources:
 - Open Task Manager.
 - Find explorer.exe, right-click, and select **Restart**.
3. **SOLIDWORKS Explorer:** This tool integrates directly with PDM and must be closed to avoid conflicts.
4. **SOLIDWORKS Application:** If SOLIDWORKS is running, it can lock PDM files and consume PDM resources. Close it completely.
5. **Other Third-Party Add-Ins:** Any application or add-in that integrates with PDM (e.g., custom tools) should be stopped.

TIP 40: Understanding How SOLIDWORKS PDM Task Add-Ins are launched and ran

In SOLIDWORKS PDM, add-ins are typically loaded within the context of the application where the user is logged in, such as **Windows Explorer**. These add-ins are triggered and executed directly within the application environment. However, tasks operate differently, and their unique execution flow can make them challenging to debug.

Task Execution in SOLIDWORKS PDM

Tasks are an exception to the standard add-in loading mechanism. They are executed in separate processes, making their behavior unique:

1. TaskLaunch Hook:

- The TaskLaunch hook is called when a task is queued and launched.
- It runs within a separate process called [TaskLauncher.exe](#).
- At this stage, the task is prepared and initiated.

2. TaskRun Hook:

- The TaskRun hook executes the actual task logic.
- It runs in a separate process called [TaskExecutor.exe](#).
- This isolation ensures that the task runs independently of other PDM sessions, reducing the risk of interference but increasing debugging complexity.

Challenges with Task Debugging

- Since TaskLauncher.exe and TaskExecutor.exe run outside the primary application process, debugging tasks is harder than debugging standard add-ins.
- Breakpoints set in task logic may not trigger unless properly attached to these processes during execution.



TIP 41: Debugging SOLIDWORKS PDM Add-Ins vs. Task Hooks

Debugging SOLIDWORKS PDM add-ins and task hooks requires different approaches due to how they are executed in the PDM environment. Here's how you can debug each effectively:

Debugging Add-Ins

1. Use the Add-In Debug Section:

- Open the PDM Administration Tool.
- Expand the Add-ins node.
- Right-click your add-ins and select Debug add-ins.
- Add the path to your compiled add-in DLL.
- This allows you to debug your add-in when actions are triggered in the vault.

2. Attach to the Host Process:

- You can attach your Visual Studio debugger to the host process (e.g., [Explorer.exe](#) or [SldWorks.exe](#)).
- Set breakpoints in your code, and they will trigger when your add-in executes.

Debugging Task Hooks

Task hooks, such as [TaskLaunch](#) and [TaskRun](#), run in separate processes ([TaskLauncher.exe](#) and [TaskExecutor.exe](#)), making traditional add-in debugging methods ineffective.

1. **Use the [AttachDebugger](#) Method:** Include a method like the following in your task hook to launch the debugger during execution:

```

public void AttachDebugger()
{
    Process process = Process.GetCurrentProcess();
    if (!Debugger.IsAttached)
    {
        var information = new StringBuilder();
        information.AppendLine();
        information.AppendLine($"Host      Name = {process.ProcessName}");
        information.AppendLine($"Host      Id   = {process.Id}");
        information.AppendLine($"Command Type = {poCmd.meCmdType.ToString()}");
        information.AppendLine($"Command Id   = {poCmd.mlCmdID.ToString()}");
        information.AppendLine($"Parent Handle= {poCmd.mlParentWnd}");
        information.AppendLine($"Comments      = {poCmd.mbsComment}");

        if (MessageBox.Show($"Attach Debugger? {information.ToString()}", "Debugger",
            MessageBoxButtons.OKCancel) == DialogResult.OK)
            Debugger.Launch();
    }
}

```

2. How It Works:

- This method checks if a debugger is already attached to the current process.
- If not, it prompts you with a message box asking if you want to attach the debugger.
- When you click OK, it launches the Visual Studio debugger and attaches it to the process.

3. Call [AttachDebugger](#) in Your Code:

- Add a call to [AttachDebugger](#) in the [TaskLaunch](#) or [TaskRun](#) hook.
- When the hook executes, the debugger prompt will appear.

TIP 42: Before you migrate data to SOLIDWORKS PDM

When working with files that require changes like renaming, updating custom properties, or performing bulk edits, it is often more efficient to make these changes **outside the PDM vault** before moving the files into it. This practice minimizes the computational expense associated with check-in and check-out operations:

Why Modify Outside the Vault?

1. **Reduced Overhead:** Check-in and check-out operations in SOLIDWORKS PDM involve communication with the PDM server, which can be time-consuming for large files or bulk edits.
2. **Batch Processing:** Performing changes outside the vault allows for quicker batch edits using external tools or scripts.
3. **Avoids Conflict:** Working outside the vault prevents locking conflicts or permissions issues that might occur when files are actively managed within the PDM system.

Best Practices

1. **Prepare Files in a Local Folder:** Keep a designated local folder for pre-vault processing where you can freely modify file properties, names, and structures.
2. **Use Automation Tools:** Use the SOLIDWORKS Document Manager API to edit custom properties without having to open the files.
3. **Validate Before Moving:** Double-check file references and properties to ensure compatibility with PDM requirements.

TIP 43: Use Version-Free Variables for Version-Independent Data

When working with SOLIDWORKS PDM, it's wise to use **version-free variables** for data that does not need to be tied to specific file versions. Version-free variables are independent of file versions, making them ideal for metadata or information that remains consistent across all versions of a file.

Using version-free variables in SOLIDWORKS PDM API operations significantly enhances performance because they eliminate the need for costly check-in and check-out operations. Version-free variables allow direct updates to metadata without creating new file versions, making API workflows faster and more efficient.

TIP 44: Using `IEdmBatchUnlock2` for Efficient Batch File Operations

When working with large sets of files in SOLIDWORKS PDM, performing individual operations like unlocking files one at a time can be slow and resource-intensive. The `IEdmBatchUnlock2` interface provides a more efficient way to unlock multiple files in a single operation. Here's a code example in C#:

```
using EdmLib;
using System;

public class PdmBatchUnlocker
{
    public void BatchUnlockFiles(string vaultName, string
assemblyFilePath)
    {
        try
        {
            // Initialize the vault object
            IEdmVault5 vault = new EdmVault5();
            if (!vault.IsLoggedIn)
            {
                vault.LoginAuto(vaultName, 0);
            }

            if (string.IsNullOrEmpty(assemblyFilePath))
            {
                throw new ArgumentException("Assembly file path
cannot be null or empty.");
            }

            // Retrieve the assembly file
            IEdmFile5 assemblyFile =
vault.GetFileFromPath(assemblyFilePath, out IEdmFolder5
parentFolder);
            if (assemblyFile == null)
```

```

        {
            throw new InvalidOperationException($"The file
'{{assemblyFilePath}}' was not found in the vault.");
        }

        // Initialize the batch unlocker utility
        IEdmBatchUnlock2 batchUnlocker =
        (IEdmBatchUnlock2) vault.CreateUtility(EdmUtility.EdmUtil_BatchUnl
        ock);

        // Prepare the selection list for the batch unlocker
        var selection = new EdmSelItem
        {
            mlDocID = assemblyFile.ID,
            mlProjID = parentFolder.ID
        };

        batchUnlocker.AddSelection((EdmVault5) vault, ref
        selection);

        // Build the unlock tree and set additional options
        batchUnlocker.CreateTree(0,
        (int)EdmUnlockBuildTreeFlags.Eubtf_ShowCloseAfterCheckinOption +
        (int)EdmUnlockBuildTreeFlags.Eubtf_MayUnlock);
        batchUnlocker.Comment = "Batch unlock operation";

        // Show the unlock dialog (optional, can be skipped
        for silent execution)
        // bool dialogResult = batchUnlocker.ShowDialog();

        // Unlock files
        batchUnlocker.UnlockFiles(0, null);

        // Verify statuses after unlock
        object statuses =
        batchUnlocker.GetStatus((int)EdmUnlockStatusFlag.Eusf_CloseAfterC
        heckinFlag);
        Console.WriteLine("Batch unlock operation completed.
        Status: " + statuses);
    }
    catch (System.Runtime.InteropServices.COMException ex)
    {
        Console.WriteLine("HRESULT = 0x" +
        ex.ErrorCode.ToString("X") + " " + ex.Message);
    }
}

```

```
    }  
    catch (Exception ex)  
    {  
        Console.WriteLine("Error: " + ex.Message);  
    }  
}  
}
```



TIP 45: Retrieving the Last Check-in Time of a File Using *IEdmHistory* in SOLIDWORKS PDM

The *IEdmHistory* interface in SOLIDWORKS PDM provides access to the history of a file, including check-in and check-out events. You can use this to determine the last time a file was checked out and by whom.

Here is a code sample that returns the last time a file is checked out.

Here is a code sample that returns the last time a file is checked out.

```
public void GetLastCheckoutTime(IEdmVault5 vault, string
filePath)
{
    try
    {
        // Get the file and its parent folder
        IEdmFile5 file = vault.GetFileFromPath(filePath, out _);
        if (file == null)
        {
            throw new InvalidOperationException($"File not found:
{filePath}");
        }

        // Get the file history
        IedmHistory3 history =
history = (IEdmHistory3)vault2.CreateUtility(EdmUtility.EdmUtil_H
istory);
        history.AddFile(file.ID);
    }
}
```

```

        EdmHistoryItem[] ppoRethistory = null;
        history.GetSortedHistory(ref ppoRethistory, (int)
EdmHistoryType.Edmhist_FileVersion);

        var lastChecked = string.empty;

        var latestItem = historyItems .Where(item =>
item.meType ==
EdmHistoryType.Edmhist_FileVersion).OrderByDescending(item =>
DateTime.Parse(item.moDate)).FirstOrDefault();

        // Retrieve the lastChecked value
        lastChecked = latestItem?.moDate;

Console.WriteLine($"Last Checked: {lastChecked}");

    }
    catch (System.Runtime.InteropServices.COMException ex)
    {
        Console.WriteLine("HRESULT = 0x" +
ex.ErrorCode.ToString("X") + " " + ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }
}

```

TIP 46: Check for Local Copy

Before Accessing Files in SOLIDWORKS PDM

When working with SOLIDWORKS PDM API, it's better to check if a file has a local copy before downloading a local copy to the vault.

Here's a utility method to determine the state of a local copy for a given file:

Here's a utility method to determine the state of a local copy for a given file:

```
public static LocalCopyState HasLocalCopy(this IEdmFile5 file)
{
    bool pbLocalOverwrittenVersionObsolete = false;

    // Get the current version of the file
    var currentVersion = file.CurrentVersion;

    // Get the local version using IEdmFile12
    var localVersion = (file as
    IEdmFile12).GetLocalVersionNo2(file.GetParentFolderID(), out
    pbLocalOverwrittenVersionObsolete);

    // Determine the local copy state
    if (localVersion == -1)
        return LocalCopyState.ThereIsNoLocalCopy;
    else
    {
        if (pbLocalOverwrittenVersionObsolete)
            return
            LocalCopyState.LocalCopyObsoleteServerHasNewOne;
        else
        {
            if (localVersion == currentVersion)
                return LocalCopyState.LocalAndServerCopySynced;
            else
```

```
        return LocalCopyState.LocalCopyIsBehindServer;  
    }  
}  
}
```



TIP 47: Minimize Dependencies in PDM Add-Ins to Avoid Assembly Resolution Issues

When developing SOLIDWORKS PDM add-ins, remember that the add-ins are loaded into the **AppDomain** of the host application, such as [ConisioAdmin.exe](#), [explorer.exe](#), or any custom standalone application that interacts with the vault. This can lead to **assembly resolution issues**, especially if your add-in has multiple dependencies or conflicting versions of libraries with other add-ins that are already loaded.

TIP 48: Ideal Number of Add-ins

Per SOLIDWORKS PDM Vault

Your add-ins consist of DLL files. Each time an application like `Explorer.exe` logs into PDM, it loads those files into the application's session, increasing startup time and memory usage. While memory use is not a major issue these days, startup time can be noticeably affected.

The ideal number of add-ins is five, but PDM already includes three default ones:

1. **Dispatch** – Automates workflows via scripting.
2. **SOLIDWORKS Task Add-in** – Add-in used to run custom SOLIDWORKS macros (Convert, Print, Designer Checker...).
3. **SOLIDWORKS PDF Task Add-in** – Add-in used to convert Office and DraftSight documents to PDF.

This means **you only have 2 slots left for custom add-ins** before performance starts to degrade.

Each add-in adds **overhead**, increasing **vault load times**. Instead of multiple separate add-ins, **merge functionalities into one** to keep performance smooth.

- ✓ **PDM already has 3 built-in add-ins**, leaving space for just **2 custom add-ins**.
- ✓ Too many add-ins slow down vault operations.
- ✓ **Merging multiple add-ins into one** improves performance & simplifies maintenance.

TIP 49: Sign All Assemblies and Use Signed NuGet Packages to Avoid Assembly Resolution Issues

When developing SOLIDWORKS PDM add-ins, ensuring all your assemblies are **signed** and using **signed NuGet packages** is critical to avoiding assembly resolution issues. Strong-named assemblies are uniquely identified by their name, version, culture, and public key token. This ensures that the correct version of the dependency is loaded, even if other assemblies in the AppDomain reference different versions.

TIP 50: Use Logging to Debug and Monitor SOLIDWORKS PDM Add-Ins

In addition to using traditional file-based logging systems, you can use the [IEdmDictionary5](#) class to store logs in memory for quick access during runtime. This approach is particularly useful for scenarios where logs need to be analyzed dynamically or when file-based logging is not practical.

TIP 51: Where to Look for Information and Help About the SOLIDWORKS PDM API

Finding reliable information and help for the SOLIDWORKS PDM API is crucial for efficient development. Here are the best resources and strategies to get assistance:

1. Official SOLIDWORKS API Documentation

- **Location:**
 - Installed locally with the SOLIDWORKS PDM client.
- **Details:**
 - Provides comprehensive documentation, including methods, properties, enumerations, and examples.
 - Covers versions starting from your installed SOLIDWORKS PDM release.

2. SOLIDWORKS Forums ([Hosted on the 3DEXperience platform now](#))

- **Location:** SOLIDWORKS Forum
- **Why Use It:**
 - Engaged community of professionals and developers.
 - Answers from experienced users and SOLIDWORKS employees.
 - Discussions on real-world use cases and solutions.
- **Tip:** Search the forum first to check if your issue has already been addressed.

3. CADOverflow

- **Location:** [CADOverflow](#)
- **Why Use It:**
 - A dedicated Q&A site for CAD and PDM-related topics.
 - Focused community of CAD and PDM developers.
 - Specific, targeted answers for technical API questions.

4. SOLIDWORKS Resellers and Technical Support

- **Why Use Them:**
 - Resellers often have experts available to answer technical API questions.
 - Use your SOLIDWORKS subscription benefits to get support directly from SOLIDWORKS.