jesseaknight - 1

Automation title : **In a multi-body part, create a set of configurations - one for each body. Create a Keep/Delete bodies for each, and keep only the one body.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
public class KeepOnlyMacro
{
    public static object GenerateKeepOnlyConfigurations()
    {
        try
        {
            Console.WriteLine("[START] Beginning comprehensive KeepOnly features and configurations creation macro");

            // Create SolidWorks instance (will be fixed by executor to use Marshal.GetActiveObject)
            SldWorks swApp = Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application" };
            }

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a multi-body part document." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Check if the active document is a part
            if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
```

```
        {
            return new { success = false, error = "Active document is not a part. Please open a
multi-body part document." };
        }

        IPartDoc swPart = swModel as IPartDoc;
        if (swPart == null)
        {
            return new { success = false, error = "Could not cast to IPartDoc interface" };
        }

        Console.WriteLine("[INFO] Part document confirmed");

        // Get all solid bodies
        object[] bodies = swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as
object[];

        if (bodies == null || bodies.Length == 0)
        {
            return new { success = false, error = "No solid bodies found in the part" };
        }

        if (bodies.Length == 1)
        {
            return new { success = false, error = "Only one solid body found. This macro is
designed for multi-body parts with multiple solid bodies." };
        }

        Console.WriteLine("[INFO] Found " + bodies.Length + " solid bodies in the part");

        // Get active configuration name to use as base
        IConfigurationManager swConfigMgr = swModel.ConfigurationManager;
        IConfiguration swActiveConfig = swConfigMgr.ActiveConfiguration;
        string baseConfigName = swActiveConfig.Name;

        Console.WriteLine("[INFO] Base configuration: " + baseConfigName);

        int configurationsCreated = 0;
        int keepOnlyFeaturesCreated = 0;
        System.Collections.Generic.List<string> createdConfigs = new
System.Collections.Generic.List<string>();
        System.Collections.Generic.List<string> createdFeatures = new
System.Collections.Generic.List<string>();

        // Process each body and create both KeepOnly feature AND configuration for it
        for (int i = 0; i < bodies.Length; i++)
        {
            IBody2 swBody = bodies[i] as IBody2;
```

```
            if (swBody == null)
            {
                Console.WriteLine("[WARNING] Could not cast body " + (i + 1) + " to IBody2");
                continue;
            }

            string bodyName = swBody.Name;
            Console.WriteLine("[INFO] Processing body " + (i + 1) + " of " + bodies.Length + ":
'" + bodyName + "'");

            // Handle empty or null body names by creating a default name
            if (string.IsNullOrEmpty(bodyName) || string.IsNullOrWhiteSpace(bodyName))
            {
                bodyName = "Body" + (i + 1);
                Console.WriteLine("[INFO] Body had empty name, using default: " +
bodyName);
            }

            // Create configuration name (sanitize body name for config name)
            string configName = "Config_" + bodyName.Replace(" ", "").Replace("-",
"").Replace(".", "_").Replace("[", "").Replace("]", "");
            string keepOnlyFeatureName = "KeepOnly_" + bodyName.Replace("[",
"").Replace("]", "");

            Console.WriteLine("[DEBUG] Creating configuration: " + configName);
            Console.WriteLine("[DEBUG] Creating KeepOnly feature: " +
keepOnlyFeatureName);

            try
            {
                // Check if configuration already exists
                string[] existingConfigs = swModel.GetConfigurationNames() as string[];
                bool configExists = false;
                if (existingConfigs != null)
                {
                    foreach (string existingConfig in existingConfigs)
                    {
                        if (existingConfig == configName)
                        {
                            configExists = true;
                            break;
                        }
                    }
                }

                IConfiguration swBodyConfig = null;

                if (configExists)
```

```
            {
                Console.WriteLine("[INFO] Configuration " + configName + " already exists,
will update it");
            }
            else
            {
                // Create new configuration based on the active configuration
                swBodyConfig = swConfigMgr.AddConfiguration2(
                    configName,                                     // Configuration name
                    "Configuration showing only " + bodyName,       //
Description/comment
                    "",                                             // Alternate name
                    (int)swConfigurationOptions2_e.swConfigOption_DontActivate,   // Don't
activate immediately
                    baseConfigName,                                 // Parent configuration
                    "",                                             // Suppress new feature mode
                    false                                           // Link to parent BOM
                );

                if (swBodyConfig == null)
                {
                    Console.WriteLine("[ERROR] Failed to create configuration for body: " +
bodyName);
                    continue;
                }

                Console.WriteLine("[SUCCESS] Created configuration: " + configName);
                configurationsCreated++;
                createdConfigs.Add(configName);
            }

            // Activate the configuration to work in it
            Console.WriteLine("[DEBUG] Activating configuration: " + configName);
            bool activated = swModel.ShowConfiguration2(configName);

            if (!activated)
            {
                Console.WriteLine("[ERROR] Failed to activate configuration: " +
configName);
                continue;
            }

            Console.WriteLine("[DEBUG] Configuration activated successfully");

            // Clear any existing selections
            swModel.ClearSelection2(true);

            // Create array of all OTHER bodies (bodies to delete, keeping only current one)
```

```
            System.Collections.Generic.List<IBody2> bodiesToDelete = new
System.Collections.Generic.List<IBody2>();

            for (int j = 0; j < bodies.Length; j++)
            {
                if (j != i) // Skip the body we want to keep
                {
                    IBody2 bodyToDelete = bodies[j] as IBody2;
                    if (bodyToDelete != null)
                    {
                        bodiesToDelete.Add(bodyToDelete);
                    }
                }
            }

            Console.WriteLine("[DEBUG] Will delete " + bodiesToDelete.Count + " bodies,
keeping: " + bodyName);

            if (bodiesToDelete.Count > 0)
            {
                // Select the bodies to delete
                ISelectionMgr swSelMgr = swModel.ISelectionManager;

                // Convert list to array for selection
                IBody2[] deleteArray = bodiesToDelete.ToArray();

                Console.WriteLine("[DEBUG] Selecting " + deleteArray.Length + " bodies for
deletion");

                // Select all bodies to delete
                int selectedCount = swModel.Extension.MultiSelect2(deleteArray, false, null);

                if (selectedCount == deleteArray.Length)
                {
                    Console.WriteLine("[DEBUG] Successfully selected " + selectedCount + "
bodies");

                    // Create Delete Body feature (false = delete selected bodies, which acts
as KeepOnly for remaining body)
                    IFeature swDeleteBodyFeature =
swModel.FeatureManager.InsertDeleteBody2(false);

                    if (swDeleteBodyFeature != null)
                    {
                        // Rename the feature for clarity - this is our KeepOnly feature
                        swDeleteBodyFeature.Name = keepOnlyFeatureName;
```

```
                        Console.WriteLine("[SUCCESS] Created KeepOnly feature: " +
swDeleteBodyFeature.Name);
                        keepOnlyFeaturesCreated++;
                        createdFeatures.Add(keepOnlyFeatureName);

                        // Suppress/unsuppress
                        string[] baseConfigArray = new string[] { baseConfigName };
                        swDeleteBodyFeature.SetSuppression2(
                            (int)swFeatureSuppressionAction_e.swSuppressFeature,
                            (int)swInConfigurationOpts_e.swSpecifyConfiguration,
                            baseConfigArray
                        );

                        string[] bodyConfigArray = new string[] { configName };
                        swDeleteBodyFeature.SetSuppression2(
                            (int)swFeatureSuppressionAction_e.swUnSuppressFeature,
                            (int)swInConfigurationOpts_e.swSpecifyConfiguration,
                            bodyConfigArray
                        );

                        Console.WriteLine("[SUCCESS] Completed setup for " + configName + "
with KeepOnly feature " + keepOnlyFeatureName + " (showing only " + bodyName + ")");
                    }
                    else
                    {
                        Console.WriteLine("[ERROR] Failed to create KeepOnly (Delete Body)
feature for " + bodyName);
                    }
                }
                else
                {
                    Console.WriteLine("[ERROR] Could only select " + selectedCount + " out
of " + deleteArray.Length + " bodies");
                }

                // Clear selections
                swModel.ClearSelection2(true);
            }
            else
            {
                Console.WriteLine("[INFO] No other bodies to delete for " + bodyName + "
(single body case)");
                if (!configExists)
                {
                    configurationsCreated++;
                    createdConfigs.Add(configName);
                }
            }
```

```
            }
            catch (Exception bodyEx)
            {
                Console.WriteLine("[ERROR] Exception processing body " + bodyName + ": " +
bodyEx.Message);
            }
        }

        // Return to the base configuration
        Console.WriteLine("[INFO] Returning to base configuration: " + baseConfigName);
        swModel.ShowConfiguration2(baseConfigName);

        // Rebuild the model
        swModel.ForceRebuild3(false);
        swModel.GraphicsRedraw2();

        Console.WriteLine("[COMPLETE] Macro execution finished");
        Console.WriteLine("[RESULTS] Successfully created " + configurationsCreated + "
configurations and " + keepOnlyFeaturesCreated + " KeepOnly features out of " +
bodies.Length + " bodies");

        return new {
            success = true,
            message = "Created " + configurationsCreated + " configurations and " +
keepOnlyFeaturesCreated + " KeepOnly features",
            totalBodies = bodies.Length,
            configurationsCreated = configurationsCreated,
            keepOnlyFeaturesCreated = keepOnlyFeaturesCreated,
            configurationNames = createdConfigs,
            keepOnlyFeatureNames = createdFeatures,
            baseConfiguration = baseConfigName
        };
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Main exception: " + ex.Message);
        Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
        return new { success = false, error = "Exception: " + ex.Message };
    }
  }
}
```

chris-b-co

Automation title : **Add a different colour to each solid body within a multi body part**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
public class Main
{
    private static System.Random random = new System.Random();

    public static object Execute()
    {
        try
        {
            Console.WriteLine("[START] Beginning multi-body color assignment macro");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application"
};
            }

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a part
document." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Check if the active document is a part
            if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
            {
```

```
            return new { success = false, error = "Active document is not a part. Please open a part document with multiple solid bodies." };
        }

        IPartDoc swPart = swModel as IPartDoc;
        if (swPart == null)
        {
            return new { success = false, error = "Could not cast to IPartDoc" };
        }

        Console.WriteLine("[INFO] Part document confirmed");

        // Get all solid bodies
        object[] bodies = swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as object[];

        if (bodies == null || bodies.Length == 0)
        {
            return new { success = false, error = "No solid bodies found in the part" };
        }

        if (bodies.Length == 1)
        {
            return new { success = false, error = "Only one solid body found. This macro is designed for multi-body parts." };
        }

        Console.WriteLine("[INFO] Found " + bodies.Length + " solid bodies in the part");

        // Define distinct colors for different bodies
        double[][] predefinedColors = new double[][]
        {
            // Red
            new double[] { 0.8, 0.2, 0.2, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Green
            new double[] { 0.2, 0.8, 0.2, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Blue
            new double[] { 0.2, 0.2, 0.8, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Orange
            new double[] { 1.0, 0.5, 0.0, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Purple
            new double[] { 0.8, 0.2, 0.8, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Yellow
            new double[] { 1.0, 1.0, 0.2, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Cyan
            new double[] { 0.2, 0.8, 0.8, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 },
            // Pink
            new double[] { 1.0, 0.4, 0.6, 1.0, 1.0, 0.5, 0.3125, 0.0, 0.0 }
```

```
        };

        int colorizedBodies = 0;
        int totalFacesColored = 0;

        // Apply different colors to each body by coloring all its faces
        for (int i = 0; i < bodies.Length; i++)
        {
            IBody2 swBody = bodies[i] as IBody2;
            if (swBody == null) continue;

            Console.WriteLine("[INFO] Processing body " + (i + 1) + " of " + bodies.Length);

            try
            {
                // Get color for this body (cycle through predefined colors if we have more
bodies than colors)
                double[] matProps = predefinedColors[i % predefinedColors.Length];

                Console.WriteLine("[DEBUG] Getting faces for body " + (i + 1));

                // Get all faces of the body
                object[] faces = swBody.GetFaces() as object[];

                if (faces != null && faces.Length > 0)
                {
                    Console.WriteLine("[DEBUG] Found " + faces.Length + " faces in body " + (i
+ 1));

                    int facesColoredForThisBody = 0;

                    // Apply the same color to all faces of this body
                    foreach (object faceObj in faces)
                    {
                        IFace2 swFace = faceObj as IFace2;
                        if (swFace != null)
                        {
                            try
                            {
                                // Apply color to face using the pattern from user macro
                                swFace.SetMaterialPropertyValues2(matProps,
                                    (int)swInConfigurationOpts_e.swThisConfiguration,
                                    null);

                                facesColoredForThisBody++;
                                totalFacesColored++;
                            }
                            catch (Exception faceEx)
```

```
                    {
                        Console.WriteLine("[ERROR] Failed to apply color to face: " +
faceEx.Message);
                    }
                }
            }

            if (facesColoredForThisBody > 0)
            {
                colorizedBodies++;
                Console.WriteLine("[SUCCESS] Applied color to body " + (i + 1) + " (" +
facesColoredForThisBody + " faces)");
            }
            else
            {
                Console.WriteLine("[WARNING] No faces were colored for body " + (i +
1));
            }
        }
        else
        {
            Console.WriteLine("[WARNING] No faces found for body " + (i + 1));
        }
    }
    catch (Exception bodyEx)
    {
        Console.WriteLine("[ERROR] Failed to process body " + (i + 1) + ": " +
bodyEx.Message);
    }
}

// Refresh graphics to show changes
Console.WriteLine("[INFO] Refreshing graphics to display changes");
swModel.GraphicsRedraw2();

Console.WriteLine("[COMPLETE] Macro execution finished");
Console.WriteLine("[RESULTS] Successfully colored " + colorizedBodies + " out of "
+ bodies.Length + " solid bodies (" + totalFacesColored + " total faces)");

return new {
    success = true,
    message = "Successfully applied different colors to " + colorizedBodies + " solid
bodies in the multi-body part (" + totalFacesColored + " faces colored)",
    bodiesProcessed = colorizedBodies,
    totalBodies = bodies.Length,
    facesColored = totalFacesColored
};
}
```

```
        catch (Exception ex)
        {
            Console.WriteLine("[ERROR] Exception occurred: " + ex.Message);
            return new { success = false, error = "Exception: " + ex.Message };
        }
    }
}
```

goofballtech

Automation title : **Bulk Save As configurations to step/stl using the config name. I have to apply each config then file - save as for each one of the config versions and type the right variant.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swconst;
using System;

public class Export
{
    public static object ExportAllConfigurations()
    {
        try
        {
            Console.WriteLine("[START] Beginning bulk configuration export macro (STEP & STL)");

            // Create SolidWorks instance (will be fixed by executor to use Marshal.GetActiveObject)
            SldWorks swApp = Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application" };
            }

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a part or assembly document." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());
```

```
        // Check if document is saved
        string currentPath = swModel.GetPathName();
        if (string.IsNullOrEmpty(currentPath))
        {
            return new { success = false, error = "Please save the document before running
this macro." };
        }

        Console.WriteLine("[INFO] Document path: " + currentPath);

        // Get directory and base filename
        string directory = System.IO.Path.GetDirectoryName(currentPath);
        string fileNameWithoutExt =
System.IO.Path.GetFileNameWithoutExtension(currentPath);

        Console.WriteLine("[INFO] Export directory: " + directory);
        Console.WriteLine("[INFO] Base filename: " + fileNameWithoutExt);

        // Get configuration manager and current configuration
        IConfigurationManager swConfigMgr = swModel.ConfigurationManager;
        IConfiguration swActiveConfig = swConfigMgr.ActiveConfiguration;
        string originalConfigName = swActiveConfig.Name;

        Console.WriteLine("[INFO] Original configuration: " + originalConfigName);

        // Get all configuration names
        string[] configNames = swModel.GetConfigurationNames() as string[];

        if (configNames == null || configNames.Length == 0)
        {
            return new { success = false, error = "No configurations found in the document." };
        }

        Console.WriteLine("[INFO] Found " + configNames.Length + " configuration(s): " +
string.Join(", ", configNames));

        int successfulStepExports = 0;
        int successfulStlExports = 0;
        int totalConfigs = configNames.Length;
        System.Collections.Generic.List<string> exportedFiles = new
System.Collections.Generic.List<string>();
        System.Collections.Generic.List<string> errorMessages = new
System.Collections.Generic.List<string>();

        // Export each configuration
        for (int i = 0; i < configNames.Length; i++)
        {
```

```
            string configName = configNames[i];
            Console.WriteLine("[INFO] Processing configuration " + (i + 1) + " of " +
totalConfigs + ": '" + configName + "'");

            try
            {
                // Activate the configuration
                Console.WriteLine("[DEBUG] Activating configuration: " + configName);
                bool activated = swModel.ShowConfiguration2(configName);

                if (!activated)
                {
                    string errorMsg = "Failed to activate configuration: " + configName;
                    Console.WriteLine("[ERROR] " + errorMsg);
                    errorMessages.Add(errorMsg);
                    continue;
                }

                Console.WriteLine("[SUCCESS] Configuration activated: " + configName);

                // Clean up config name for filename
                string safeConfigName = configName
                    .Replace(" ", "_")
                    .Replace("/", "_")
                    .Replace("\\", "_")
                    .Replace(":", "_")
                    .Replace("*", "_")
                    .Replace("?", "_")
                    .Replace("\"", "_")
                    .Replace("<", "_")
                    .Replace(">", "_")
                    .Replace("|", "_");

                // Export STEP
                Console.WriteLine("[DEBUG] Exporting configuration '" + configName + "' to
STEP...");
                string stepFileName = fileNameWithoutExt + "_" + safeConfigName + ".step";
                string stepFilePath = System.IO.Path.Combine(directory, stepFileName);

                swModel.ClearSelection2(true);

                int stepErrors = 0;
                int stepWarnings = 0;

                bool stepResult = swModel.Extension.SaveAs(
                    stepFilePath,
                    (int)swSaveAsVersion_e.swSaveAsCurrentVersion,
                    (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
```

```
                    null,
                    ref stepErrors,
                    ref stepWarnings
                );

                if (stepResult && System.IO.File.Exists(stepFilePath))
                {
                    Console.WriteLine("[SUCCESS] STEP export successful: " + stepFileName);
                    successfulStepExports++;
                    exportedFiles.Add(stepFileName);
                }
                else
                {
                    string errorMsg = "Failed to export STEP for config '" + configName + "'.
Errors: " + stepErrors + ", Warnings: " + stepWarnings;
                    Console.WriteLine("[ERROR] " + errorMsg);
                    errorMessages.Add(errorMsg);
                }

                // Export STL (only for parts)
                if (swModel.GetType() == (int)swDocumentTypes_e.swDocPART)
                {
                    Console.WriteLine("[DEBUG] Exporting configuration '" + configName + "' to
STL...");
                    string stlFileName = fileNameWithoutExt + "_" + safeConfigName + ".stl";
                    string stlFilePath = System.IO.Path.Combine(directory, stlFileName);

                    swModel.ClearSelection2(true);

                    int stlErrors = 0;
                    int stlWarnings = 0;

                    bool stlResult = swModel.Extension.SaveAs(
                        stlFilePath,
                        (int)swSaveAsVersion_e.swSaveAsCurrentVersion,
                        (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
                        null,
                        ref stlErrors,
                        ref stlWarnings
                    );

                    if (stlResult && System.IO.File.Exists(stlFilePath))
                    {
                        Console.WriteLine("[SUCCESS] STL export successful: " + stlFileName);
                        successfulStlExports++;
                        exportedFiles.Add(stlFileName);
                    }
                    else
```

```
                {
                    string errorMsg = "Failed to export STL for config '" + configName + "'.
Errors: " + stlErrors + ", Warnings: " + stlWarnings;
                    Console.WriteLine("[ERROR] " + errorMsg);
                    errorMessages.Add(errorMsg);
                }
            }
            else
            {
                Console.WriteLine("[INFO] Skipping STL export for assemblies");
            }
        }
        catch (System.Exception configEx)
        {
            string errorMsg = "Exception processing config '" + configName + "': " +
configEx.Message;
            Console.WriteLine("[ERROR] " + errorMsg);
            errorMessages.Add(errorMsg);
        }
    }

    // Return to original configuration
    Console.WriteLine("[INFO] Returning to original configuration: " +
originalConfigName);
    swModel.ShowConfiguration2(originalConfigName);

    Console.WriteLine("[COMPLETE] Bulk export finished");
    Console.WriteLine("[RESULTS] Exported " + successfulStepExports + " STEP and "
+ successfulStlExports + " STL (if applicable) out of " + totalConfigs + " configurations");

    // Build result message
    string resultMessage = "Exported " + totalConfigs + " configurations:\n";
    resultMessage += "- " + successfulStepExports + " STEP files\n";

    if (swModel.GetType() == (int)swDocumentTypes_e.swDocPART)
        resultMessage += "- " + successfulStlExports + " STL files\n";
    else
        resultMessage += "- STL skipped (assemblies not supported)\n";

    resultMessage += "\nFiles saved to: " + directory;

    if (errorMessages.Count > 0)
        resultMessage += "\n\nWarning: " + errorMessages.Count + " export(s) had
issues.";

    return new {
        success = true,
        message = resultMessage,
```

```
        totalConfigurations = totalConfigs,
        stepExports = successfulStepExports,
        stlExports = successfulStlExports,
        exportDirectory = directory,
        exportedFiles = exportedFiles,
        errors = errorMessages,
        originalConfiguration = originalConfigName
    };
}
catch (System.Exception ex)
{
    Console.WriteLine("[ERROR] Main exception: " + ex.Message);
    Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
    return new { success = false, error = "Exception: " + ex.Message };
}
```

melon_zest

Automation title : **Input XYZ points or a csv file and make a 3D sketch of those points. Always wished that was a thing. Maybe it is and I haven't looked hard enough.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swconst;
using System;

public class Create3D
{
    public static object Create3DSketchFromPoints()
    {
        try
        {
            Console.WriteLine("[START] Beginning XYZ points input and 3D sketch creation
macro");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
                return new { success = false, error = "Could not connect to SolidWorks application"
};

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;
            if (swModel == null)
                return new { success = false, error = "No document is open. Please open a part
document." };

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Check if the active document is a part
            if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
                return new { success = false, error = "Active document is not a part. Please open a
part document." };
```

```
        Console.WriteLine("[INFO] Part document confirmed");

        // Input dialog for XYZ coords
        Console.WriteLine("[INFO] Showing input dialog for XYZ coordinates");
        string coordInput = ShowCoordinateInputDialog();
        if (string.IsNullOrEmpty(coordInput))
            return new { success = false, error = "No coordinates provided", cancelled = true };

        Console.WriteLine("[INFO] User provided coordinates: " + coordInput);

        // Parse coords
        var coordinates = ParseCoordinates(coordInput);
        if (coordinates.Count == 0)
            return new { success = false, error = "No valid coordinates found. Format: X,Y,Z
per line" };

        Console.WriteLine("[INFO] Parsed " + coordinates.Count + " coordinate sets");

        // Get Sketch Manager
        ISketchManager swSketchMgr = swModel.SketchManager;
        if (swSketchMgr == null)
            return new { success = false, error = "Could not get sketch manager" };

        swModel.ClearSelection2(true);
        Console.WriteLine("[INFO] Creating 3D sketch");
        swSketchMgr.Insert3DSketch(true);
        swSketchMgr.AddToDB = true;

        int pointsCreated = 0;
        foreach (var coord in coordinates)
        {
            ISketchPoint swSketchPoint = swSketchMgr.CreatePoint(coord[0], coord[1],
coord[2]);
            if (swSketchPoint != null) pointsCreated++;
        }

        swSketchMgr.AddToDB = false;
        swSketchMgr.Insert3DSketch(true);

        swModel.ForceRebuild3(false);
        swModel.GraphicsRedraw2();

        Console.WriteLine("[SUCCESS] 3D sketch with " + pointsCreated + " points created
successfully");

        return new {
            success = true,
            message = "Created 3D sketch with " + pointsCreated + " points",
```

```
            pointsCreated = pointsCreated,
            totalCoordinates = coordinates.Count
        };
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Exception occurred: " + ex.Message);
        return new { success = false, error = "Exception: " + ex.Message };
    }
}

private static string ShowCoordinateInputDialog()
{
    try
    {
        var form = new System.Windows.Forms.Form
        {
            Text = "XYZ Coordinates Input",
            Width = 500,
            Height = 400,
            StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen,
            FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog,
            MaximizeBox = false,
            MinimizeBox = false
        };

        var instructionLabel = new System.Windows.Forms.Label
        {
            Text = "Enter XYZ coordinates (one set per line, comma-separated):",
            Left = 10, Top = 10, Width = 460, Height = 20
        };
        var exampleLabel = new System.Windows.Forms.Label
        {
            Text = "Example: 0,0,0 or 10.5,20.3,5.7 (coordinates assumed in mm)",
            Left = 10, Top = 35, Width = 460, Height = 20
        };
        var textBox = new System.Windows.Forms.TextBox
        {
            Left = 10, Top = 65, Width = 460, Height = 220,
            Multiline = true, ScrollBars = System.Windows.Forms.ScrollBars.Both,
            Text = "0,0,0" + Environment.NewLine + "10,0,0" + Environment.NewLine +
"0,10,0" + Environment.NewLine + "0,0,10"
        };

        var okButton = new System.Windows.Forms.Button
        {
            Text = "Create 3D Sketch", Left = 310, Top = 300, Width = 120, Height = 30,
            DialogResult = System.Windows.Forms.DialogResult.OK
```

```
            };
            var cancelButton = new System.Windows.Forms.Button
            {
                Text = "Cancel", Left = 440, Top = 300, Width = 75, Height = 30,
                DialogResult = System.Windows.Forms.DialogResult.Cancel
            };

            form.Controls.Add(instructionLabel);
            form.Controls.Add(exampleLabel);
            form.Controls.Add(textBox);
            form.Controls.Add(okButton);
            form.Controls.Add(cancelButton);
            form.AcceptButton = okButton;
            form.CancelButton = cancelButton;

            return form.ShowDialog() == System.Windows.Forms.DialogResult.OK ?
textBox.Text.Trim() : string.Empty;
        }
        catch
        {
            return string.Empty;
        }
    }

    private static System.Collections.Generic.List<double[]> ParseCoordinates(string input)
    {
        var coordinates = new System.Collections.Generic.List<double[]>();
        string[] lines = input.Split(new[] { Environment.NewLine },
StringSplitOptions.RemoveEmptyEntries);

        foreach (string line in lines)
        {
            string[] parts = line.Split(new[] { ',', ';', ' ' }, StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length >= 3)
            {
                try
                {
                    double x = double.Parse(parts[0].Trim()) / 1000.0;
                    double y = double.Parse(parts[1].Trim()) / 1000.0;
                    double z = double.Parse(parts[2].Trim()) / 1000.0;
                    coordinates.Add(new[] { x, y, z });
                }
                catch { }
            }
        }
        return coordinates;
    }
}
```

LightFeisty1809

Automation title : **Link Sheet metal flat pattern o/a sizes to custom properties ex. Plate 1000 x 200 x 1,2**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class SheetMetalDimension
{
    public static object StartLogic()
    {
        try
        {
            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =

System.Runtime.InteropServices.Marshal.GetActiveObject("SldWorks.Application")
                as SldWorks;

            if (swApp == null)
            {
                return new
                {
                    success = false,
                    error = "Could not connect to SolidWorks application",
                };
            }

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new
                {
                    success = false,
                    error = "No document is open. Please open a sheet metal part.",
                };
            }

            // Check if the active document is a part
```

```
        if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
        {
            return new
            {
                success = false,
                error = "Active document must be a part. Please open a sheet metal part
document.",
            };
        }

        // Cast to part document
        IPartDoc swPart = swModel as IPartDoc;
        if (swPart == null)
        {
            return new { success = false, error = "Failed to cast to part document" };
        }

        // Check if this is a sheet metal part
        bool isSheetMetal = CheckIfSheetMetalPart(swModel);
        if (!isSheetMetal)
        {
            return new
            {
                success = false,
                error = "This is not a sheet metal part. Please open a sheet metal part with flat
pattern feature.",
            };
        }

        // Look for existing flat pattern or unfold feature
        IFeature flatPatternFeature = FindFlatPatternOrUnfoldFeature(swModel);

        if (flatPatternFeature != null)
        {
            // Check if feature is currently suppressed
            bool isSuppressed = flatPatternFeature.IsSuppressed();

            if (isSuppressed)
            {
                // Unsuppress the feature to show unfolded view
                bool result = flatPatternFeature.SetSuppression2(
                    (int)swFeatureSuppressionAction_e.swUnSuppressFeature,
                    (int)swInConfigurationOpts_e.swThisConfiguration,
                    null
                );

                if (!result)
                {
```

```
            return new
            {
                success = false,
                error = "Failed to unsuppress flat pattern feature",
            };
        }
    }
}
else
{
    // No flat pattern feature exists - create one
    swModel.ClearSelection2(true);
    SelectSheetMetalFacesAndBends(swModel);

    try
    {
        swModel.InsertSheetMetalUnfold();
        flatPatternFeature = FindFlatPatternOrUnfoldFeature(swModel);
    }
    catch (Exception unfoldEx)
    {
        return new
        {
            success = false,
            error = "Failed to create unfold feature: " + unfoldEx.Message,
        };
    }
}

// Get flat pattern dimensions
double[] dimensions = GetFlatPatternDimensions(swModel, flatPatternFeature);
if (dimensions == null || dimensions.Length != 3)
{
    return new { success = false, error = "Failed to get flat pattern dimensions" };
}

// Fold the part back
if (flatPatternFeature != null && !flatPatternFeature.IsSuppressed())
{
    flatPatternFeature.SetSuppression2(
        (int)swFeatureSuppressionAction_e.swSuppressFeature,
        (int)swInConfigurationOpts_e.swThisConfiguration,
        null
    );
}

// Convert dimensions from meters to mm
double length = System.Math.Round(dimensions[0] * 1000, 1);
```

```
        double width = System.Math.Round(dimensions[1] * 1000, 1);
        double thickness = System.Math.Round(dimensions[2] * 1000, 1);

        // Create custom property string
        string plateDescription =
            "Plate "
            + length.ToString("0.#").Replace(".", ",")
            + " x "
            + width.ToString("0.#").Replace(".", ",")
            + " x "
            + thickness.ToString("0.#").Replace(".", ",");

        // Get configuration and property manager
        IConfiguration swConfig = swModel.GetActiveConfiguration() as IConfiguration;
        ICustomPropertyManager customPropMgr = null;

        if (swConfig != null)
        {
            customPropMgr = swConfig.CustomPropertyManager as
ICustomPropertyManager;
        }

        if (customPropMgr == null)
        {
            customPropMgr =
                swModel.Extension.CustomPropertyManager[""] as ICustomPropertyManager;
            if (customPropMgr == null)
            {
                return new { success = false, error = "Could not get custom property manager"
};
            }
        }

        // Add/Update custom properties
        string[] existingProps = customPropMgr.GetNames() as string[];
        bool propertyExists = false;

        if (existingProps != null)
        {
            foreach (string propName in existingProps)
            {
                if (propName == "test flat size name")
                {
                    propertyExists = true;
                    break;
                }
            }
        }
```

```
int result1;
if (propertyExists)
{
    result1 = customPropMgr.Set2("test flat size name", plateDescription);
}
else
{
    result1 = customPropMgr.Add3(
        "test flat size name",
        (int)swCustomInfoType_e.swCustomInfoText,
        plateDescription,
        (int)swCustomPropertyAddOption_e.swCustomPropertyReplaceValue
    );
}

// Add individual dimension properties
customPropMgr.Add3(
    "SM_Length",
    (int)swCustomInfoType_e.swCustomInfoText,
    length.ToString("0.#"),
    (int)swCustomPropertyAddOption_e.swCustomPropertyReplaceValue
);
customPropMgr.Add3(
    "SM_Width",
    (int)swCustomInfoType_e.swCustomInfoText,
    width.ToString("0.#"),
    (int)swCustomPropertyAddOption_e.swCustomPropertyReplaceValue
);
customPropMgr.Add3(
    "SM_Thickness",
    (int)swCustomInfoType_e.swCustomInfoText,
    thickness.ToString("0.#"),
    (int)swCustomPropertyAddOption_e.swCustomPropertyReplaceValue
);

if (result1 >= 0)
{
    // Refresh views
    try
    {
        swModel.GraphicsRedraw2();
        swModel.FeatureManager.UpdateFeatureTree();
        swModel.ViewDisplayShaded();
    }
    catch { }

    // Release COM objects
```

```
            try
            {
                if (customPropMgr != null)

System.Runtime.InteropServices.Marshal.ReleaseComObject(customPropMgr);
                if (swConfig != null)
                    System.Runtime.InteropServices.Marshal.ReleaseComObject(swConfig);
                if (swPart != null)
                    System.Runtime.InteropServices.Marshal.ReleaseComObject(swPart);
            }
            catch { }

            return new
            {
                success = true,
                message = "Successfully linked flat pattern dimensions to custom properties",
                plateDescription = plateDescription,
                length = length,
                width = width,
                thickness = thickness,
            };
        }
        else
        {
            return new { success = false, error = "Failed to add custom property" };
        }
    }
    catch (Exception ex)
    {
        return new { success = false, error = "Exception: " + ex.Message };
    }
}

// Helper method: Check if sheet metal part
private static bool CheckIfSheetMetalPart(IModelDoc2 swModel)
{
    try
    {
        IFeatureManager swFeatMgr = swModel.FeatureManager as IFeatureManager;
        if (swFeatMgr == null)
            return false;

        IFeature swFeature = swModel.FirstFeature() as IFeature;
        while (swFeature != null)
        {
            string featType = swFeature.GetTypeName2();
            if (
                featType.Contains("SheetMetal")
```

```
                || featType.Equals("BaseFlange")
                || featType.Equals("EdgeFlange")
                || featType.Equals("FlatPattern")
                || featType.Equals("ProcessBends")
            )
            {
                return true;
            }
            swFeature = swFeature.GetNextFeature() as IFeature;
        }
        return false;
    }
    catch
    {
        return false;
    }
}

// Helper method: Find flat pattern or unfold feature
private static IFeature FindFlatPatternOrUnfoldFeature(IModelDoc2 swModel)
{
    try
    {
        IFeature swFeature = swModel.FirstFeature() as IFeature;
        while (swFeature != null)
        {
            string featType = swFeature.GetTypeName2();
            string featName = swFeature.Name;
            if (
                featType.Equals("FlatPattern")
                || featType.Equals("Unfold")
                || featName.Contains("Flat-Pattern")
                || featName.Contains("Unfold")
            )
            {
                return swFeature;
            }
            swFeature = swFeature.GetNextFeature() as IFeature;
        }
        return null;
    }
    catch
    {
        return null;
    }
}

// Helper method: Select sheet metal faces and bends
```

```
    private static bool SelectSheetMetalFacesAndBends(IModelDoc2 swModel)
    {
        try
        {
            IPartDoc swPart = swModel as IPartDoc;
            if (swPart == null)
                return false;

            object[] bodies = swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as
object[];
            if (bodies == null || bodies.Length == 0)
                return false;

            foreach (object bodyObj in bodies)
            {
                IBody2 swBody = bodyObj as IBody2;
                if (swBody == null)
                    continue;

                object[] faces = swBody.GetFaces() as object[];
                if (faces != null && faces.Length > 0)
                {
                    IFace2 swFace = faces[0] as IFace2;
                    if (swFace != null)
                    {
                        IEntity faceEntity = swFace as IEntity;
                        if (faceEntity != null)
                        {
                            if (faceEntity.Select4(false, null))
                            {
                                return true;
                            }
                        }
                    }
                }
            }
            return false;
        }
        catch
        {
            return false;
        }
    }

    // Helper method: Get flat pattern dimensions
    private static double[] GetFlatPatternDimensions(
        IModelDoc2 swModel,
        IFeature flatPatternFeature
```

```
    )
    {
        try
        {
            IPartDoc swPart = swModel as IPartDoc;
            if (swPart == null)
                return null;

            object[] bodies = swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as
object[];
            if (bodies == null || bodies.Length == 0)
                return null;

            IBody2 largestBody = null;
            double largestVolume = 0.0;

            foreach (object bodyObj in bodies)
            {
                IBody2 swBody = bodyObj as IBody2;
                if (swBody == null)
                    continue;

                double[] massProps = swBody.GetMassProperties(1.0) as double[];
                if (massProps != null && massProps.Length > 0)
                {
                    double volume = massProps[3];
                    if (volume > largestVolume)
                    {
                        largestVolume = volume;
                        largestBody = swBody;
                    }
                }
            }

            if (largestBody == null)
                return null;

            double[] boundingBox = largestBody.GetBodyBox() as double[];
            if (boundingBox == null || boundingBox.Length != 6)
                return null;

            double lengthX = System.Math.Abs(boundingBox[3] - boundingBox[0]);
            double lengthY = System.Math.Abs(boundingBox[4] - boundingBox[1]);
            double lengthZ = System.Math.Abs(boundingBox[5] - boundingBox[2]);

            double[] sortedDimensions = new double[] { lengthX, lengthY, lengthZ };
            System.Array.Sort(sortedDimensions);
            System.Array.Reverse(sortedDimensions);
```

```
            return sortedDimensions;
        }
        catch
        {
            return null;
        }
    }
}
```

# Odd_knock

Automation title : **Autosave when the mouse stops moving or no keystrokes or the application is backgrounded for 25+ seconds.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class Autosave
 {
      // Import necessary Windows API functions for activity monitoring
      [System.Runtime.InteropServices.DllImport("user32.dll")]
      private static extern bool GetLastInputInfo(ref LASTINPUTINFO plii);

      [System.Runtime.InteropServices.DllImport("user32.dll")]
      private static extern IntPtr GetForegroundWindow();

      [System.Runtime.InteropServices.DllImport("user32.dll")]
      private static extern int GetWindowThreadProcessId(IntPtr hWnd, out uint processId);

      [System.Runtime.InteropServices.StructLayout(
         System.Runtime.InteropServices.LayoutKind.Sequential
      )]
      private struct LASTINPUTINFO
      {
         public uint cbSize;
         public uint dwTime;
      }

      private static System.Threading.Timer monitoringTimer;
      private static System.DateTime lastActivityTime;
      private static System.DateTime lastAutosaveTime;
      private static SldWorks swApp;
      private static bool isMonitoring = false;
      private static uint solidWorksProcessId;

      public static object StartAutosave()
      {
         try
         {
            Console.WriteLine("[START] Beginning autosave activity monitoring macro");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
```

```
swApp =
    System.Activator.CreateInstance(
        System.Type.GetTypeFromProgID("SldWorks.Application")
    ) as SldWorks;

if (swApp == null)
{
    return new
    {
        success = false,
        error = "Could not connect to SolidWorks application",
    };
}

Console.WriteLine("[INFO] Connected to SolidWorks");

// Get SolidWorks process ID for focus monitoring
IntPtr swHwnd = new IntPtr(swApp.IFrameObject().GetHWnd());
GetWindowThreadProcessId(swHwnd, out solidWorksProcessId);
Console.WriteLine("[INFO] SolidWorks Process ID: " + solidWorksProcessId);

// Check if monitoring is already running
if (isMonitoring)
{
    return new
    {
        success = true,
        message = "Autosave monitoring is already running.",
    };
}

// Start the monitoring system with timer (non-blocking)
StartMonitoring();

string message =
    "Autosave monitoring started! Will autosave when: "
    + "Mouse stops moving for 7+ seconds, "
    + "No keystrokes for 7+ seconds, or "
    + "SolidWorks is backgrounded for 7+ seconds. "
    + "Monitoring will continue until SolidWorks is closed.";

// Start an async task that runs forever to keep the UI showing "Running"
Task.Run(async () =>
{
    while (isMonitoring && swApp != null)
    {
        await Task.Delay(1000);
    }
```

```
            });

            // Keep the macro "running" by using async delay in a loop
            try
            {
                while (isMonitoring && swApp != null)
                {
                    System.Threading.Thread.Sleep(100);
                    System.Windows.Forms.Application.DoEvents(); // Process Windows
messages to prevent freezing
                }
            }
            catch (System.Threading.ThreadAbortException)
            {
                // Thread is being aborted (user clicked stop)
                Console.WriteLine("[INFO] Macro execution stopped by user");
                StopMonitoring(); // Clean up the timer
                throw; // Re-throw to let the abort complete
            }
            finally
            {
                // Always clean up when exiting
                if (isMonitoring)
                {
                    StopMonitoring();
                }
            }

            return new
            {
                success = true,
                message = "Monitoring stopped",
                monitoring = false,
            };
        }
        catch (System.Exception ex)
        {
            Console.WriteLine("[ERROR] Exception: " + ex.Message);
            return new { success = false, error = "Exception: " + ex.Message };
        }
    }

    private static void StartMonitoring()
    {
        Console.WriteLine("[INFO] Starting autosave monitoring system");

        isMonitoring = true;
        lastActivityTime = System.DateTime.Now;
```

```
        lastAutosaveTime = System.DateTime.MinValue; // Initialize to allow first autosave

        // Create timer that checks every 1 second
        monitoringTimer = new System.Threading.Timer(CheckActivityAndSave, null, 1000,
1000);

        Console.WriteLine("[SUCCESS] Monitoring system started - checking every 1
second");
    }

    private static void StopMonitoring()
    {
        Console.WriteLine("[INFO] Stopping autosave monitoring system");

        isMonitoring = false;

        if (monitoringTimer != null)
        {
            monitoringTimer.Dispose();
            monitoringTimer = null;
        }

        Console.WriteLine("[SUCCESS] Monitoring system stopped");
    }

    private static void CheckActivityAndSave(object state)
    {
        try
        {
            if (!isMonitoring || swApp == null)
            {
                return;
            }

            // Check if the COM object is still valid
            try
            {
                var test = swApp.Visible; // Quick test to see if COM object is still valid
            }
            catch
            {
                Console.WriteLine("[INFO] SolidWorks connection lost - stopping monitoring");
                StopMonitoring();
                return;
            }

            System.DateTime currentTime = System.DateTime.Now;
            bool shouldAutosave = false;
```

```
string reason = "";

// Check 1: Mouse and keyboard activity using GetLastInputInfo
LASTINPUTINFO lastInputInfo = new LASTINPUTINFO();
lastInputInfo.cbSize = (uint)
    System.Runtime.InteropServices.Marshal.SizeOf(lastInputInfo);

if (GetLastInputInfo(ref lastInputInfo))
{
    uint tickCount = (uint)System.Environment.TickCount;
    uint idleTime = tickCount - lastInputInfo.dwTime;
    double idleSeconds = idleTime / 1000.0;

    if (idleSeconds >= 7.0)
    {
        shouldAutosave = true;
        reason =
            "No mouse/keyboard activity for "
            + idleSeconds.ToString("F1")
            + " seconds";
    }
}

// Check 2: Application focus state
IntPtr foregroundWindow = GetForegroundWindow();
if (foregroundWindow != System.IntPtr.Zero)
{
    uint foregroundProcessId;
    GetWindowThreadProcessId(foregroundWindow, out foregroundProcessId);

    // Check if SolidWorks is not the active application
    if (foregroundProcessId != solidWorksProcessId)
    {
        // SolidWorks is backgrounded - check how long
        System.TimeSpan backgroundTime = currentTime - lastActivityTime;

        if (backgroundTime.TotalSeconds >= 7.0)
        {
            shouldAutosave = true;
            reason =
                "SolidWorks backgrounded for "
                + backgroundTime.TotalSeconds.ToString("F1")
                + " seconds";
        }
    }
    else
    {
        // SolidWorks is active, update last activity time
```

```
                lastActivityTime = currentTime;
            }
        }

        // Perform autosave if conditions are met
        if (shouldAutosave)
        {
            // Check cooldown period - prevent autosave spam (minimum 30 seconds
between autosaves)
            System.TimeSpan timeSinceLastAutosave = currentTime - lastAutosaveTime;
            if (timeSinceLastAutosave.TotalSeconds < 30.0)
            {
                Console.WriteLine(
                    "[COOLDOWN] Autosave skipped - only "
                        + timeSinceLastAutosave.TotalSeconds.ToString("F1")
                        + " seconds since last autosave"
                );
                return;
            }

            Console.WriteLine("[TRIGGER] Autosave triggered: " + reason);
            PerformAutosave();

            // Reset the activity timer and update last autosave time
            lastActivityTime = currentTime;
            lastAutosaveTime = currentTime;
        }
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Monitoring error: " + ex.Message);
    }
}

private static void PerformAutosave()
{
    try
    {
        Console.WriteLine("[AUTOSAVE] Starting automatic save operation...");

        // Get all open model windows - FIXED API CALL
        IFrame swFrame = swApp.Frame() as IFrame;
        if (swFrame == null)
        {
            Console.WriteLine("[ERROR] Could not get SolidWorks frame");
            return;
        }
```

```
            // Show non-intrusive status bar notification
            swFrame.SetStatusBarText("AutoSave in progress...");

            object vModelWnds = swFrame.ModelWindows;

            if (vModelWnds == null)
            {
                Console.WriteLine("[INFO] No documents open to save");
                return;
            }

            object[] modelWindows = vModelWnds as object[];
            if (modelWindows == null || modelWindows.Length == 0)
            {
                Console.WriteLine("[INFO] No model windows found");
                return;
            }

            int savedCount = 0;
            int skippedCount = 0;
            System.Collections.Generic.HashSet<string> savedDocuments =
                new System.Collections.Generic.HashSet<string>();

            Console.WriteLine("[INFO] Found " + modelWindows.Length + " open
documents");

            // First, save all open documents (including assemblies)
            for (int i = 0; i < modelWindows.Length; i++)
            {
                try
                {
                    IModelWindow swModelWnd = modelWindows[i] as IModelWindow;
                    if (swModelWnd == null)
                        continue;

                    IModelDoc2 swModel = swModelWnd.ModelDoc;
                    if (swModel == null)
                        continue;

                    // Check if document needs saving
                    if (swModel.GetSaveFlag())
                    {
                        Console.WriteLine("[SAVING] Document: " + swModel.GetTitle());

                        // Save silently without dialogs
                        int errors = 0;
                        int warnings = 0;
```

```
                    bool saveResult = swModel.Save3(
                        (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
                        ref errors,
                        ref warnings
                    );

                    if (saveResult)
                    {
                        savedCount++;
                        savedDocuments.Add(swModel.GetPathName()); // Track saved
documents
                        Console.WriteLine("[SUCCESS] Saved: " + swModel.GetTitle());
                    }
                    else
                    {
                        skippedCount++;
                        Console.WriteLine(
                            "[ERROR] Failed to save "
                                + swModel.GetTitle()
                                + " - Error: "
                                + errors
                        );
                    }
                }
                else
                {
                    Console.WriteLine("[SKIP] No changes to save: " + swModel.GetTitle());
                }
            }
            catch (System.Exception docEx)
            {
                Console.WriteLine(
                    "[ERROR] Error saving document " + i + ": " + docEx.Message
                );
                skippedCount++;
            }
        }

        // Also save all referenced documents that might have been modified in-context
        Console.WriteLine("[INFO] Checking for modified referenced documents...");
        SaveAllReferencedDocuments(ref savedCount, ref skippedCount,
savedDocuments);

        // Update status bar with results
        string statusMessage = "AutoSave: Saved " + savedCount + " document(s)";
        if (skippedCount > 0)
        {
            statusMessage += ", Failed: " + skippedCount;
```

```
            }

            swFrame.SetStatusBarText(statusMessage);
            Console.WriteLine("[COMPLETE] " + statusMessage);

            // Show completion notification
            if (savedCount > 0)
            {
                // Only show dialog when documents were actually saved - FIXED STRING
                string completionMessage = "AutoSave completed!" +
System.Environment.NewLine + System.Environment.NewLine + savedCount + "
document(s) saved successfully.";
                Console.WriteLine(
                    "[NOTIFICATION] AutoSave completed - " + savedCount + " documents
saved"
                );

                swApp.SendMsgToUser2(
                    completionMessage,
                    (int)swMessageBoxIcon_e.swMbInformation,
                    (int)swMessageBoxBtn_e.swMbOk
                );
            }
            else
            {
                // For no changes, just update status bar - no popup dialog
                Console.WriteLine(
                    "[NOTIFICATION] AutoSave completed - no documents needed saving"
                );
                swFrame.SetStatusBarText("AutoSave: No changes to save");
            }
        }
        catch (System.Exception ex)
        {
            Console.WriteLine("[ERROR] Autosave operation failed: " + ex.Message);
        }
    }

    private static void SaveAllReferencedDocuments(
        ref int savedCount,
        ref int skippedCount,
        System.Collections.Generic.HashSet<string> alreadySavedDocuments
    )
    {
        try
        {
            // Get all currently open documents in SolidWorks
            object[] openDocs = swApp.GetDocuments() as object[];
```

```
if (openDocs == null || openDocs.Length == 0)
{
    Console.WriteLine("[INFO] No additional documents found to check");
    return;
}

Console.WriteLine(
    "[INFO] Checking "
        + openDocs.Length
        + " total open documents for unsaved changes"
);

foreach (object docObj in openDocs)
{
    try
    {
        IModelDoc2 swModel = docObj as IModelDoc2;
        if (swModel == null)
            continue;

        // Skip if this document was already saved in the first phase
        string docPath = swModel.GetPathName();
        if (alreadySavedDocuments.Contains(docPath))
        {
            Console.WriteLine("[SKIP] Already saved: " + swModel.GetTitle());
            continue;
        }

        // Check if this document needs saving and wasn't already processed
        if (swModel.GetSaveFlag())
        {
            Console.WriteLine(
                "[SAVING] Referenced document: " + swModel.GetTitle()
            );

            int errors = 0;
            int warnings = 0;

            bool saveResult = swModel.Save3(
                (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
                ref errors,
                ref warnings
            );

            if (saveResult)
            {
                savedCount++;
                Console.WriteLine(
```

```
                        "[SUCCESS] Saved referenced: " + swModel.GetTitle()
                    );
                }
                else
                {
                    skippedCount++;
                    Console.WriteLine(
                        "[ERROR] Failed to save referenced "
                            + swModel.GetTitle()
                            + " - Error: "
                            + errors
                    );
                }
            }
        }
        catch (System.Exception docEx)
        {
            Console.WriteLine(
                "[ERROR] Error processing referenced document: " + docEx.Message
            );
            skippedCount++;
        }
    }
}
catch (System.Exception ex)
{
    Console.WriteLine("[ERROR] Error checking referenced documents: " +
ex.Message);
}
}
}
```

idlespoon

Automation title : **bulk opening a large selection of DXFs and saving them out as SOLIDWORKS drawings (.slddrw).**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class Bulk
{
  public static object LetsGo()
      {
         try
         {
            Console.WriteLine(
               "[START] Beginning bulk DXF to SLDDRW conversion with folder selection"
            );

            // FIX 1: Use Marshal.GetActiveObject to connect to existing SolidWorks instance
            SldWorks swApp = null;
            try
            {
               swApp = Marshal.GetActiveObject("SldWorks.Application") as SldWorks;
            }
            catch
            {
               // If no active instance, then create a new one
               swApp =
                  Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application"))
                  as SldWorks;
               if (swApp != null)
               {
                  swApp.Visible = true;
                  System.Threading.Thread.Sleep(2000); // Wait for SolidWorks to fully
initialize
               }
            }

            if (swApp == null)
            {
```

```
    return new
    {
        success = false,
        error = "Could not connect to SolidWorks application",
    };
}

Console.WriteLine("[SUCCESS] Connected to SolidWorks application");

// Show folder browser dialog to get source folder containing DXF files
Console.WriteLine("[INFO] Showing folder browser dialog");
string folderPath = ShowFolderBrowserDialog(
    "Select folder containing DXF files:",
    "DXF to SLDDRW Conversion"
);

if (string.IsNullOrEmpty(folderPath))
{
    Console.WriteLine("[INFO] User cancelled folder selection");
    return new
    {
        success = false,
        error = "No folder selected",
        cancelled = true,
    };
}

Console.WriteLine("[INFO] Selected folder: " + folderPath);

// Validate that the folder exists
if (!System.IO.Directory.Exists(folderPath))
{
    Console.WriteLine("[ERROR] Folder does not exist: " + folderPath);
    string errorMsg = "The specified folder does not exist: " + folderPath;
    swApp.SendMsgToUser2(
        errorMsg,
        (int)swMessageBoxIcon_e.swMbStop,
        (int)swMessageBoxBtn_e.swMbOk
    );
    return new { success = false, error = errorMsg };
}

// Look for DXF files in the folder
string[] dxfFiles = System.IO.Directory.GetFiles(
    folderPath,
    "*.dxf",
    System.IO.SearchOption.TopDirectoryOnly
);
```

```
            Console.WriteLine("[INFO] Found " + dxfFiles.Length + " DXF files in folder");

            if (dxfFiles.Length == 0)
            {
                string errorMsg = "No DXF files found in: " + folderPath;
                swApp.SendMsgToUser2(
                    errorMsg,
                    (int)swMessageBoxIcon_e.swMbStop,
                    (int)swMessageBoxBtn_e.swMbOk
                );
                return new { success = false, error = errorMsg };
            }

            // Show confirmation dialog with file count
            string confirmMsg =
                "Found "
                + dxfFiles.Length
                + " DXF files in the selected folder.\n\nProceed with conversion to SLDDRW
format?";
            int confirmResult = swApp.SendMsgToUser2(
                confirmMsg,
                (int)swMessageBoxIcon_e.swMbQuestion,
                (int)swMessageBoxBtn_e.swMbYesNo
            );

            if (confirmResult != (int)swMessageBoxResult_e.swMbHitYes)
            {
                Console.WriteLine("[INFO] User cancelled conversion");
                return new
                {
                    success = false,
                    error = "Conversion cancelled by user",
                    cancelled = true,
                };
            }

            // Create output directory for SLDDRW files
            string outputDir = System.IO.Path.Combine(folderPath, "SLDDRW_Output");
            if (!System.IO.Directory.Exists(outputDir))
            {
                System.IO.Directory.CreateDirectory(outputDir);
                Console.WriteLine("[INFO] Created output directory: " + outputDir);
            }

            int convertedCount = 0;
            int failedCount = 0;
            System.Collections.Generic.List<string> convertedFiles =
```

```
    new System.Collections.Generic.List<string>();
System.Collections.Generic.List<string> failedFiles =
    new System.Collections.Generic.List<string>();

// Show progress message
string progressMsg =
    "Processing "
    + dxfFiles.Length
    + " DXF files...\n\nThis may take a while. Please be patient.";
swApp.SendMsgToUser2(
    progressMsg,
    (int)swMessageBoxIcon_e.swMbInformation,
    (int)swMessageBoxBtn_e.swMbOk
);

// Process each DXF file
for (int i = 0; i < dxfFiles.Length; i++)
{
    string dxfFile = dxfFiles[i];
    string fileName = System.IO.Path.GetFileNameWithoutExtension(dxfFile);
    IImportDxfDwgData importData = null;
    IModelDoc2 newDrawing = null;

    try
    {
        Console.WriteLine(
            "[INFO] Processing DXF file "
                + (i + 1)
                + "/"
                + dxfFiles.Length
                + ": "
                + fileName
        );

        // FIX 2: Add small delay between file operations
        if (i > 0)
        {
            System.Threading.Thread.Sleep(500); // Half second delay between files
        }

        // Get DXF import data
        Console.WriteLine("[DEBUG] Getting import file data for: " + dxfFile);
        importData = swApp.GetImportFileData(dxfFile) as IImportDxfDwgData;

        if (importData == null)
        {
            Console.WriteLine("[ERROR] Failed to get import data for: " + fileName);
            failedCount++;
```

```
                    failedFiles.Add(fileName + " (Failed to get import data)");
                    continue;
                }

                Console.WriteLine("[DEBUG] Import data obtained successfully");

                // Import the DXF file directly using LoadFile4
                Console.WriteLine("[DEBUG] Loading DXF file: " + dxfFile);
                int loadErrors = 0;
                newDrawing =
                    swApp.LoadFile4(dxfFile, "", importData, ref loadErrors) as IModelDoc2;

                if (newDrawing == null)
                {
                    Console.WriteLine(
                        "[ERROR] Failed to load DXF file: "
                            + fileName
                            + " (Load errors: "
                            + loadErrors
                            + ")"
                    );
                    failedCount++;
                    failedFiles.Add(
                        fileName + " (Failed to load - Error code: " + loadErrors + ")"
                    );
                    continue;
                }

                Console.WriteLine(
                    "[SUCCESS] DXF file loaded successfully as: " + newDrawing.GetTitle()
                );

                // FIX 3: Ensure document is fully loaded
                System.Threading.Thread.Sleep(200);

                // Prepare output file path
                string slddrwPath = System.IO.Path.Combine(outputDir, fileName +
".slddrw");
                Console.WriteLine("[DEBUG] Saving as SLDDRW: " + slddrwPath);

                // Save the document as .slddrw
                int saveErrors = 0;
                int saveWarnings = 0;
                bool saveResult = newDrawing.SaveAs4(
                    slddrwPath,
                    (int)swSaveAsVersion_e.swSaveAsCurrentVersion,
                    (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
                    ref saveErrors,
```

```
            ref saveWarnings
        );

        if (!saveResult)
        {
            Console.WriteLine(
                "[ERROR] Failed to save drawing: "
                    + fileName
                    + " (Save errors: "
                    + saveErrors
                    + ", Save warnings: "
                    + saveWarnings
                    + ")"
            );
            swApp.CloseDoc(newDrawing.GetTitle());
            failedCount++;
            failedFiles.Add(
                fileName + " (Failed to save - Error: " + saveErrors + ")"
            );
            continue;
        }

        Console.WriteLine(
            "[SUCCESS] Successfully converted and saved: " + fileName + ".slddrw"
        );

        // FIX 4: Add delay before closing to ensure save is complete
        System.Threading.Thread.Sleep(200);

        // Close the drawing document
        swApp.CloseDoc(newDrawing.GetTitle());

        convertedCount++;
        convertedFiles.Add(fileName + ".slddrw");
    }
    catch (System.Exception fileEx)
    {
        Console.WriteLine(
            "[ERROR] Exception processing DXF file "
                + fileName
                + ": "
                + fileEx.Message
        );
        failedCount++;
        failedFiles.Add(fileName + " (Exception: " + fileEx.Message + ")");
    }
    finally
    {
```

```
        // FIX 5: Properly release COM objects
        if (newDrawing != null)
        {
            Marshal.ReleaseComObject(newDrawing);
            newDrawing = null;
        }
        if (importData != null)
        {
            Marshal.ReleaseComObject(importData);
            importData = null;
        }

        // FIX 6: Force garbage collection periodically
        if ((i + 1) % 10 == 0)
        {
            GC.Collect();
            GC.WaitForPendingFinalizers();
            GC.Collect();
        }
    }
}

Console.WriteLine("[COMPLETE] DXF to SLDDRW conversion finished");
Console.WriteLine(
    "[RESULTS] Converted: " + convertedCount + ", Failed: " + failedCount
);
Console.WriteLine("[INFO] SLDDRW files saved to: " + outputDir);

// Show comprehensive results to user
string resultMessage = "✅ DXF to SLDDRW Conversion Complete!\n\n";
resultMessage += "📊 Results Summary:\n";
resultMessage += "• Successfully converted: " + convertedCount + " files\n";

if (failedCount > 0)
{
    resultMessage += "• Failed to convert: " + failedCount + " files\n";
}

resultMessage += "• Total files processed: " + dxfFiles.Length + "\n\n";
resultMessage += "📁 Output Directory:\n" + outputDir + "\n\n";

if (convertedFiles.Count > 0)
{
    resultMessage += "✅ Successfully Converted Files:\n";
    for (int i = 0; i < System.Math.Min(convertedFiles.Count, 5); i++)
    {
        resultMessage += "• " + convertedFiles[i] + "\n";
    }
```

```
            if (convertedFiles.Count > 5)
            {
                resultMessage += "... and " + (convertedFiles.Count - 5) + " more files\n";
            }
        }

        if (failedFiles.Count > 0)
        {
            resultMessage += "\n⚠️ Failed Files:\n";
            for (int i = 0; i < System.Math.Min(failedFiles.Count, 3); i++)
            {
                resultMessage += "• " + failedFiles[i] + "\n";
            }
            if (failedFiles.Count > 3)
            {
                resultMessage += "... and " + (failedFiles.Count - 3) + " more failures\n";
            }
        }

        swApp.SendMsgToUser2(
            resultMessage,
            (int)swMessageBoxIcon_e.swMbInformation,
            (int)swMessageBoxBtn_e.swMbOk
        );

        return new
        {
            success = true,
            message = "Successfully converted "
                + convertedCount
                + " DXF files to SLDDRW format",
            converted = convertedCount,
            failed = failedCount,
            totalFiles = dxfFiles.Length,
            outputDirectory = outputDir,
            convertedFiles = convertedFiles.ToArray(),
            failedFiles = failedFiles.ToArray(),
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Main exception: " + ex.Message);
        string errorMsg = "Main execution error: " + ex.Message;
        try
        {
            SldWorks swApp2 =
                Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application"))
                as SldWorks;
```

```
            if (swApp2 != null)
            {
                swApp2.SendMsgToUser2(
                    errorMsg,
                    (int)swMessageBoxIcon_e.swMbStop,
                    (int)swMessageBoxBtn_e.swMbOk
                );
            }
        }
        catch
        { /* Ignore secondary error */
        }
        return new { success = false, error = errorMsg };
    }
}

// FIX 7: Ensure dialog runs on STA thread - MADE STATIC
private static string ShowFolderBrowserDialog(string description, string title)
{
    string selectedPath = string.Empty;

    System.Threading.Thread thread = new System.Threading.Thread(() =>
    {
        try
        {
            Console.WriteLine("[DEBUG] Creating folder browser dialog");

            // Create FolderBrowserDialog
            System.Windows.Forms.FolderBrowserDialog folderDialog =
                new System.Windows.Forms.FolderBrowserDialog();
            folderDialog.Description = description;
            folderDialog.ShowNewFolderButton = false;
            folderDialog.RootFolder = System.Environment.SpecialFolder.MyComputer;

            Console.WriteLine("[DEBUG] Showing folder browser dialog to user");

            // Show dialog and get result
            System.Windows.Forms.DialogResult result = folderDialog.ShowDialog();

            if (result == System.Windows.Forms.DialogResult.OK)
            {
                Console.WriteLine(
                    "[DEBUG] User selected folder: " + folderDialog.SelectedPath
                );
                selectedPath = folderDialog.SelectedPath;
            }
            else
            {
```

```
                Console.WriteLine("[DEBUG] User cancelled folder browser dialog");
            }
        }
        catch (System.Exception ex)
        {
            Console.WriteLine("[ERROR] Folder browser dialog error: " + ex.Message);
        }
    });

    thread.SetApartmentState(System.Threading.ApartmentState.STA);
    thread.Start();
    thread.Join();

    return selectedPath;
  }
}
```

Noxpertyet

Automation title : **I'd like a macro that goes through and hides the original planes and sketches if any remain turned on. Company prefers these to be turned off at the part level. Extra points if it could run though the parts in an assembly.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class HideMacro
{
    // Helper class to replace tuple for C# 7.3 compatibility
    private class DialogResultData
    {
        public bool okPressed;
        public long selectionFilter;
        public bool allConfigCheck;
    }

    // Global collections to match VBA behavior
    private static Dictionary<string, Component2> colComponents =
        new Dictionary<string, Component2>();
    private static HashSet<string> colCompList = new HashSet<string>();
    private static List<Feature> colFeatures = new List<Feature>();
    private static long selFilter = 0;
    private static bool allConfigCheck = false;

    public static object Hide()
    {
        try
        {
            Console.WriteLine("[INFO] Starting Blank Reference Features macro");

            // Show the feature selection dialog
            Console.WriteLine("[INFO] Showing feature selection dialog");
            var dialogResult = ShowFeatureSelectionDialog();

            if (!dialogResult.okPressed)
            {
                Console.WriteLine("[INFO] User cancelled operation");
                return "Operation cancelled by user";
            }
```

```
            selFilter = dialogResult.selectionFilter;
            allConfigCheck = dialogResult.allConfigCheck;

            Console.WriteLine(
                "[INFO] Selection filter: " + selFilter + ", Processing active configuration only"
            );
            // Get SolidWorks application using pure COM interop
            SldWorks swApp = null;
            try
            {
                swApp = (SldWorks)Marshal.GetActiveObject("SldWorks.Application");
            }
            catch (Exception ex)
            {
                Console.WriteLine("[ERROR] Could not connect to SolidWorks: " +
ex.Message);
                return "Could not connect to SolidWorks application";
            }

            if (swApp == null)
            {
                Console.WriteLine("[ERROR] SolidWorks application not found");
                return "SolidWorks application not found";
            }

            // Get active document
            var swModel = (ModelDoc2)swApp.ActiveDoc;
            if (swModel == null)
            {
                Console.WriteLine("[ERROR] No active document found");
                return "No active document found";
            }

            var startTime = DateTime.Now;

            // Disable graphics update for performance
            var modView = (ModelView)swModel.ActiveView;
            modView.EnableGraphicsUpdate = false;
            swModel.FeatureManager.EnableFeatureTree = false;

            // Mark command in progress to minimize UI interference
            bool prevCmd = swApp.CommandInProgress;
            swApp.CommandInProgress = true;

            try
            {
                var docType = swModel.GetType();
```

```
            Console.WriteLine("[INFO] Document type: " + docType);

            // Now do the real processing - step by step
            Console.WriteLine("[DEBUG] Starting real processing...");

            if (docType == (int)swDocumentTypes_e.swDocPART)
            {
                ProcessPartDocument(swApp, swModel, selFilter, allConfigCheck);
            }
            else if (docType == (int)swDocumentTypes_e.swDocASSEMBLY)
            {
                ProcessAssemblyDocument(swApp, swModel, selFilter, allConfigCheck);
            }
            else
            {
                Console.WriteLine("[WARNING] Unsupported document type");
                return "Unsupported document type";
            }

            Console.WriteLine("[DEBUG] Real processing completed successfully");

            var totalTime = (DateTime.Now - startTime).TotalSeconds;
            Console.WriteLine("[INFO] Operation completed in " +
totalTime.ToString("F2") + " seconds");
            MessageBox.Show(
                "Test completed in " + totalTime.ToString("F2") + " seconds",
                "Blank Reference Features Test",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information
            );

            return "Test completed successfully in " + totalTime.ToString("F2") + "
seconds";
        }
        finally
        {
            // Re-enable graphics update
            modView.EnableGraphicsUpdate = true;
            swModel.FeatureManager.EnableFeatureTree = true;

            // Reset command in progress
            swApp.CommandInProgress = prevCmd;

            // Clear global collections like VBA code does
            colCompList.Clear();
            colComponents.Clear();
            colFeatures.Clear();
        }
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine(
            "[ERROR] Exception in BlankReferenceFeaturesClass: " + ex.Message
        );
        Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
        MessageBox.Show(
            "Error: " + ex.Message,
            "Blank Reference Features Error",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error
        );
        return "Error: " + ex.Message;
    }
}

private static DialogResultData ShowFeatureSelectionDialog()
{
    Console.WriteLine("[DEBUG] Creating feature selection dialog");

    // Create the main dialog form
    var featureForm = new Form();
    featureForm.Text = "Feature Selection";
    featureForm.Width = 400;
    featureForm.Height = 480;
    featureForm.StartPosition = FormStartPosition.CenterScreen;
    featureForm.FormBorderStyle = FormBorderStyle.FixedDialog;
    featureForm.MaximizeBox = false;
    featureForm.MinimizeBox = false;

    // Add title label
    var titleLabel = new Label();
    titleLabel.Text = "Select Feature Types to Blank";
    titleLabel.Left = 20;
    titleLabel.Top = 15;
    titleLabel.Width = 350;
    titleLabel.Height = 25;
    // Remove font assignment due to C# 7.3 compatibility

    // Feature type checkboxes
    var checkBoxes = new Dictionary<string, CheckBox>();
    var featureTypes = new Dictionary<string, long>
    {
        { "Origin Profile Features", 1 },
        { "Reference Planes", 2 },
        { "Reference Axes", 4 },
        { "Reference Points", 8 },
```

```
    { "Coordinate Systems", 16 },
    { "2D Sketches", 32 },
    { "3D Sketches", 64 },
    { "3D Spline Curves", 128 },
    { "Composite Curves", 256 },
    { "Helix Features", 512 },
};

int yPos = 50;
foreach (var featureType in featureTypes)
{
    var checkBox = new CheckBox();
    checkBox.Text = featureType.Key;
    checkBox.Left = 30;
    checkBox.Top = yPos;
    checkBox.Width = 300;
    checkBox.Height = 25;
    checkBox.Checked = true; // Default to checked
    checkBoxes[featureType.Key] = checkBox;
    featureForm.Controls.Add(checkBox);
    yPos += 30;
}

// OK button
var okButton = new Button();
okButton.Text = "OK";
okButton.Left = 140;
okButton.Top = yPos + 30;
okButton.Width = 80;
okButton.Height = 35;
okButton.DialogResult = DialogResult.OK;

// Cancel button
var cancelButton = new Button();
cancelButton.Text = "Cancel";
cancelButton.Left = 240;
cancelButton.Top = yPos + 30;
cancelButton.Width = 80;
cancelButton.Height = 35;
cancelButton.DialogResult = DialogResult.Cancel;

// Add controls to form
featureForm.Controls.Add(titleLabel);
featureForm.Controls.Add(okButton);
featureForm.Controls.Add(cancelButton);

// Set default buttons
featureForm.AcceptButton = okButton;
```

```
            featureForm.CancelButton = cancelButton;

            Console.WriteLine("[DEBUG] Showing feature selection dialog to user");
            var result = featureForm.ShowDialog();

            var dialogRes = new DialogResultData();

            if (result == DialogResult.OK)
            {
                long selectionFilter = 0;
                foreach (var kvp in featureTypes)
                {
                    if (checkBoxes[kvp.Key].Checked)
                    {
                        selectionFilter |= kvp.Value;
                    }
                }

                Console.WriteLine(
                    "[INFO] Dialog OK - Filter: " + selectionFilter + ", Active config only"
                );
                dialogRes.okPressed = true;
                dialogRes.selectionFilter = selectionFilter;
                dialogRes.allConfigCheck = false; // Always false for allConfigCheck
            }
            else
            {
                Console.WriteLine("[INFO] Dialog cancelled");
                dialogRes.okPressed = false;
                dialogRes.selectionFilter = 0;
                dialogRes.allConfigCheck = false;
            }

            return dialogRes;
        }

        private static void ProcessPartDocumentSimple(SldWorks swApp, ModelDoc2
swModel)
        {
            Console.WriteLine("[DEBUG] Processing part document - simple version");

            // Clear global collections
            colFeatures.Clear();

            // Simple feature traversal - no complex loops
            var swFeat = (Feature)swModel.FirstFeature();
            int featureCount = 0;
```

```
            while (swFeat != null && featureCount < 100) // Safety limit
            {
                featureCount++;
                Console.WriteLine(
                    "[DEBUG] Processing feature " + featureCount + ": " + swFeat.Name + " (" +
swFeat.GetTypeName() + ")"
                );

                if (
                    TestFeatureType(swFeat)
                    && swFeat.Name != "Annotations"
                    && (
                        swFeat.Visible == (int)swVisibilityState_e.swVisibilityStateShown
                        || swFeat.Visible == (int)swVisibilityState_e.swVisibilityStateUnknown
                    )
                )
                {
                    colFeatures.Add(swFeat);
                    Console.WriteLine("[DEBUG] Added feature: " + swFeat.Name);
                }

                swFeat = (Feature)swFeat.GetNextFeature();
            }

            Console.WriteLine("[DEBUG] Found " + colFeatures.Count + " features to blank");

            if (colFeatures.Count > 0)
            {
                Console.WriteLine(
                    "[DEBUG] Blanking " + colFeatures.Count + " features using WORKING VBA
approach:"
                );

                try
                {
                    // VBA approach: Select all features with append=true, then blank all at once
                    int counter = 0;
                    foreach (var feature in colFeatures)
                    {
                        counter++;
                        Console.WriteLine("[DEBUG] Selecting feature " + counter + ": " +
feature.Name);

                        // VBA uses Select2(True, 0) - append to selection
                        bool selected = feature.Select2(true, 0);
                        if (!selected)
                        {
                            Console.WriteLine("[WARNING] Failed to select " + feature.Name);
```

```
                    }

                    // VBA blanks every 25 features
                    if (counter % 25 == 0 || counter == colFeatures.Count)
                    {
                        Console.WriteLine("[DEBUG] Blanking batch at " + counter + " features");
                        swModel.BlankRefGeom();
                        swModel.BlankSketch();
                        Console.WriteLine("[DEBUG] Batch " + counter + " blanked
successfully");
                    }
                }

                Console.WriteLine(
                    "[DEBUG] Successfully blanked all " + colFeatures.Count + " features!"
                );
            }
            catch (Exception ex)
            {
                Console.WriteLine("[ERROR] Exception during VBA-style blanking: " +
ex.Message);
                Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
            }
        }
    }

    private static void ProcessAssemblyDocumentSimple(SldWorks swApp, ModelDoc2
swModel)
    {
        Console.WriteLine("[DEBUG] Processing assembly document - simple version");

        // For now, just process the assembly features like a part
        ProcessPartDocumentSimple(swApp, swModel);
    }

    private static void ProcessPartDocument(
        SldWorks swApp,
        ModelDoc2 swModel,
        long selFilter,
        bool allConfigCheck
    )
    {
        Console.WriteLine("[INFO] Processing part document - active configuration only");

        // Clear global collections
        colFeatures.Clear();
        colComponents.Clear();
        colCompList.Clear();
```

```
    // Process only the active configuration
    TraverseModelFeatures(swApp, swModel, colFeatures, 0);
    BlankFeaturesInCollection(colFeatures, swModel);
}

private static void ProcessAssemblyDocument(
    SldWorks swApp,
    ModelDoc2 swModel,
    long selFilter,
    bool allConfigCheck
)
{
    Console.WriteLine("[INFO] Processing assembly document");

    // Clear global collections
    colFeatures.Clear();
    colComponents.Clear();
    colCompList.Clear();

    // Process only the active configuration
    var swConf = (Configuration)swModel.GetActiveConfiguration();
    var swRootComp = (Component2)swConf.GetRootComponent();
    AssemblyTraverseProcess(swApp, swModel, swRootComp, swConf);

    // Final blanking of features like in VBA main()
    BlankFeaturesInCollection(colFeatures, swModel);
}

private static void AssemblyTraverseProcess(
    SldWorks swApp,
    ModelDoc2 swModel,
    Component2 swRootComp,
    Configuration swConf
)
{
    swRootComp = (Component2)swConf.GetRootComponent();

    Console.WriteLine("[INFO] Processing assembly features");
    colFeatures.Clear();
    TraverseModelFeatures(swApp, swModel, colFeatures, 1);
    BlankFeaturesInCollection(colFeatures, swModel);

    Console.WriteLine("[INFO] Traversing components");
    TraverseComponent(swApp, swModel, swRootComp, 1);

    Console.WriteLine("[INFO] Processing " + colComponents.Count + " components");
    colFeatures.Clear(); // EmptyCollection colFeatures - like VBA
```

```
            foreach (var comp in colComponents.Values)
            {
                var compModelDoc = (ModelDoc2)comp.GetModelDoc();
                if (compModelDoc != null)
                {
                    TraverseComponentFeatures(swApp, compModelDoc, comp, colFeatures, 1);
                }
            }
        }

        private static void TraverseComponent(
            SldWorks swApp,
            ModelDoc2 swModel,
            Component2 swComp,
            int nLevel
        )
        {
            var vChildComp = (object[])swComp.GetChildren();
            if (vChildComp == null)
                return;

            foreach (Component2 swChildComp in vChildComp)
            {
                var componentTag = GetComponentTag(swChildComp);

                if (!colCompList.Contains(componentTag) &&
!string.IsNullOrEmpty(componentTag))
                {
                    colComponents[componentTag] = swChildComp;
                    colCompList.Add(componentTag);
                    Console.WriteLine(
                        "[DEBUG] Added component: " +
GetFileName(GetCompModelDoc(swChildComp))
                    );
                }

                TraverseComponent(swApp, swModel, swChildComp, nLevel + 1);
            }
        }

        private static void TraverseModelFeatures(
            SldWorks swApp,
            ModelDoc2 swModel,
            List<Feature> featCol,
            int nLevel
        )
        {
```

```
    var swFeat = (Feature)swModel.FirstFeature();
    TraverseFeatureFeatures(swApp, swModel, swFeat, featCol, nLevel);
}

private static void TraverseComponentFeatures(
    SldWorks swApp,
    ModelDoc2 swModel,
    Component2 swComp,
    List<Feature> featCol,
    int nLevel
)
{
    var swFeat = (Feature)swComp.FirstFeature();
    TraverseFeatureFeatures(swApp, swModel, swFeat, featCol, nLevel);
}

private static void TraverseFeatureFeatures(
    SldWorks swApp,
    ModelDoc2 swModel,
    Feature swFeat,
    List<Feature> featCol,
    int nLevel
)
{
    if (swApp == null || swModel == null || swFeat == null)
        return;

    try
    {
        // Simple approach - just traverse main features, skip sub-features for now
        while (swFeat != null)
        {
            try
            {
                if (swFeat.Name != "Annotations" && swFeat.Name != null)
                {
                    if (
                        TestFeatureType(swFeat)
                        && (
                            swFeat.Visible
                                == (int)swVisibilityState_e.swVisibilityStateShown
                            || swFeat.Visible
                                == (int)swVisibilityState_e.swVisibilityStateUnknown
                        )
                    )
                    {
                        featCol.Add(swFeat);
                        Console.WriteLine(
```

```
                                "[DEBUG] Added feature: " + swFeat.Name + " (" +
swFeat.GetTypeName() + ")"
                            );
                        }
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine("[WARNING] Error processing feature: " + ex.Message);
                }

                try
                {
                    swFeat = (Feature)swFeat.GetNextFeature();
                }
                catch (Exception ex)
                {
                    Console.WriteLine("[WARNING] Error getting next feature: " +
ex.Message);
                    break;
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("[ERROR] Exception in TraverseFeatureFeatures: " +
ex.Message);
        }
    }

    private static bool TestFeatureType(Feature feature)
    {
        var featureType = feature.GetTypeName();
        var featureValue = GetFeatureTypeValue(featureType);
        return (selFilter & featureValue) != 0;
    }

    private static long GetFeatureTypeValue(string typeName)
    {
        switch (typeName)
        {
            case "OriginProfileFeature":
                return 1;
            case "RefPlane":
                return 2;
            case "RefAxis":
                return 4;
            case "RefPoint":
```

```
                return 8;
            case "CoordSys":
                return 16;
            case "ProfileFeature":
                return 32; // 2D sketch
            case "3DProfileFeature":
                return 64; // 3D sketch
            case "3DSplineCurve":
                return 128; // curve thru points
            case "CompositeCurve":
                return 256;
            case "Helix":
                return 512;
            default:
                return 0;
        }
    }

    private static void BlankFeaturesInCollectionSimple(
        List<Feature> features,
        ModelDoc2 swModel
    )
    {
        if (features.Count == 0 || swModel == null)
            return;

        Console.WriteLine("[DEBUG] Blanking " + features.Count + " features - simple
version");

        try
        {
            // Clear any existing selections
            swModel.ClearSelection2(true);

            // Select all features - FIRST ONE IS NOT APPEND!
            bool append = false; // VBA starts with append = false!
            int counter = 0;

            foreach (var feature in features)
            {
                counter++;
                Console.WriteLine(
                    "[DEBUG] About to select feature " + counter + ": " + feature.Name + "
(append=" + append + ")"
                );
                bool selected = feature.Select2(append, 0);
                Console.WriteLine("[DEBUG] Selection completed: " + selected);
```

```
                if (!selected)
                {
                    Console.WriteLine("[WARNING] Failed to select feature: " + feature.Name);
                }

                append = true; // After first selection, append = true

                // VBA blanks every 25 features
                if (counter % 25 == 0)
                {
                    Console.WriteLine("[DEBUG] Blanking batch at " + counter);
                    swModel.BlankRefGeom();
                    swModel.BlankSketch();
                    append = false; // Reset append after blanking
                }
            }

            // Blank remaining features (if not already blanked in batch)
            if (counter % 25 != 0) // Only blank if we haven't just blanked
            {
                Console.WriteLine("[DEBUG] Final blanking of remaining features...");
                Console.WriteLine("[DEBUG] About to call BlankRefGeom...");
                swModel.BlankRefGeom();
                Console.WriteLine("[DEBUG] BlankRefGeom completed");

                Console.WriteLine("[DEBUG] About to call BlankSketch...");
                swModel.BlankSketch();
                Console.WriteLine("[DEBUG] BlankSketch completed");
            }

            // Clear selection - VBA does this by selecting and deselecting first feature
            Console.WriteLine("[DEBUG] Clearing selection...");
            if (features.Count > 0)
            {
                features[0].Select2(false, 0);
                features[0].DeSelect();
            }
            Console.WriteLine("[DEBUG] Selection cleared");

            Console.WriteLine("[DEBUG] Successfully blanked " + features.Count + "
features");
        }
        catch (Exception ex)
        {
            Console.WriteLine(
                "[ERROR] Exception in BlankFeaturesInCollectionSimple: " + ex.Message
            );
            try
```

```
            {
                swModel.ClearSelection2(true);
            }
            catch { }
        }
    }

    private static void BlankFeaturesInCollection(List<Feature> features, ModelDoc2
swModel)
    {
        if (features.Count == 0 || swModel == null)
            return;

        Console.WriteLine(
            "[INFO] Blanking " + features.Count + " features using WORKING VBA
approach"
        );

        try
        {
            // VBA approach: Select all features with append=true, then blank in batches
            int counter = 0;
            foreach (var feature in features)
            {
                counter++;
                Console.WriteLine("[DEBUG] Selecting feature " + counter + ": " +
feature.Name);

                // VBA uses Select2(True, 0) - append to selection (NO ClearSelection2!)
                bool selected = feature.Select2(true, 0);
                if (!selected)
                {
                    Console.WriteLine("[WARNING] Failed to select " + feature.Name);
                }

                // VBA blanks every 25 features
                if (counter % 25 == 0 || counter == features.Count)
                {
                    Console.WriteLine("[DEBUG] Blanking batch at " + counter + " features");
                    swModel.BlankRefGeom();
                    swModel.BlankSketch();
                    Console.WriteLine("[DEBUG] Batch " + counter + " blanked successfully");
                }
            }

            Console.WriteLine("[INFO] Successfully blanked all " + features.Count + "
features!");
        }
```

```
            catch (Exception ex)
            {
                Console.WriteLine("[ERROR] Exception in BlankFeaturesInCollection: " +
ex.Message);
                Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
            }
        }

        // Helper methods - Fixed null-conditional operator for C# 7.3 compatibility
        private static ModelDoc2 GetCompModelDoc(Component2 comp)
        {
            if (comp == null)
                return null;
            return comp.GetModelDoc() as ModelDoc2;
        }

        private static string GetComponentTag(Component2 comp)
        {
            if (comp == null)
                return "";
            var modelDoc = GetCompModelDoc(comp);
            return GetFileName(modelDoc) + "||" + comp.ReferencedConfiguration;
        }

        private static string GetFileName(ModelDoc2 doc)
        {
            if (doc == null)
                return "";
            var fullPath = doc.GetPathName();
            if (string.IsNullOrEmpty(fullPath))
                return "";

            return System.IO.Path.GetFileName(fullPath);
        }
    }
```

functi0nxy macro 1

Automation title : **I want a shortcut to select all tangent edges**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class Selector
{
    public static object SelectTangentEdges()
    {
        try
        {
            Console.WriteLine("[START] Beginning tangent edge selection macro - KEEPING
ORIGINAL EDGE SELECTED");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application"
};
            }

            Console.WriteLine("[INFO] Connected to SolidWorks");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a part,
assembly, or drawing." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Get the selection manager
```

```
        ISelectionMgr swSelMgr = swModel.ISelectionManager;

        if (swSelMgr == null)
        {
            return new { success = false, error = "Could not get selection manager" };
        }

        Console.WriteLine("[INFO] Got selection manager");

        // Check if exactly one edge is selected
        int selectedCount = swSelMgr.GetSelectedObjectCount2(-1);
        Console.WriteLine("[DEBUG] Selected objects count: " + selectedCount);

        if (selectedCount != 1)
        {
            return new { success = false, error = "Please select exactly one edge. Currently
selected: " + selectedCount + " objects." };
        }

        // Check if the selected object is an edge
        int selType = swSelMgr.GetSelectedObjectType3(1, -1);
        Console.WriteLine("[DEBUG] Selected object type: " + selType + " (swSelEDGES = "
+ (int)swSelectType_e.swSelEDGES + ")");

        if (selType != (int)swSelectType_e.swSelEDGES)
        {
            return new { success = false, error = "Please select an edge. The currently
selected object is not an edge." };
        }

        // Get the selected edge
        IEdge swSelectedEdge = swSelMgr.GetSelectedObject6(1, -1) as IEdge;

        if (swSelectedEdge == null)
        {
            return new { success = false, error = "Could not get the selected edge" };
        }

        Console.WriteLine("[INFO] Successfully got the selected edge - KEEPING IT
SELECTED");

        // Get all tangent edges using the SolidWorks API method
        Console.WriteLine("[DEBUG] Getting tangent edges...");
        object tangentEdgesObj = swSelectedEdge.GetTangentEdges();

        if (tangentEdgesObj == null)
        {
            Console.WriteLine("[INFO] No tangent edges found for the selected edge");
```

```
            return new { success = true, message = "The selected edge has no tangent edges
to add to selection. Original edge remains selected.", tangentEdgesCount = 0 };
        }

        // Cast to array of edges
        object[] tangentEdgesArray = tangentEdgesObj as object[];

        if (tangentEdgesArray == null || tangentEdgesArray.Length == 0)
        {
            Console.WriteLine("[INFO] No tangent edges found (array is null or empty)");
            return new { success = true, message = "The selected edge has no tangent edges
to add to selection. Original edge remains selected.", tangentEdgesCount = 0 };
        }

        Console.WriteLine("[INFO] Found " + tangentEdgesArray.Length + " tangent edges");

        // DO NOT clear selection - we want to keep the original edge selected!
        Console.WriteLine("[DEBUG] NOT clearing selection - keeping original edge
selected");

        // Add tangent edges to the existing selection using append=true
        int successCount = 0;
        for (int i = 0; i < tangentEdgesArray.Length; i++)
        {
            IEdge tangentEdge = tangentEdgesArray[i] as IEdge;
            if (tangentEdge != null)
            {
                Console.WriteLine("[DEBUG] Adding tangent edge " + (i + 1) + " to selection...");

                IEntity edgeEntity = tangentEdge as IEntity;
                if (edgeEntity != null)
                {
                    // Use Select2 with append = true to ADD to existing selection (not replace)
                    bool selected = edgeEntity.Select2(true, -1);
                    if (selected)
                    {
                        successCount++;
                        Console.WriteLine("[DEBUG] Successfully added tangent edge " + (i + 1) +
" to selection");
                    }
                    else
                    {
                        Console.WriteLine("[WARNING] Failed to add tangent edge " + (i + 1) + "
to selection");
                    }
                }
                else
                {
```

```
                    Console.WriteLine("[WARNING] Could not cast tangent edge " + (i + 1) + " to IEntity");
                }
            }
            else
            {
                Console.WriteLine("[WARNING] Tangent edge " + (i + 1) + " is null");
            }
        }

        // Verify final selection
        int finalSelectedCount = swSelMgr.GetSelectedObjectCount2(-1);
        Console.WriteLine("[DEBUG] Final selection count: " + finalSelectedCount);

        Console.WriteLine("[COMPLETE] Selection process finished");

        if (successCount > 0)
        {
            string message = "✅ Successfully added " + successCount + " tangent edges to your selection! Your original edge is still selected along with " + successCount + " tangent edges (total selected: " + finalSelectedCount + " edges).";
            Console.WriteLine("[SUCCESS] " + message);

            return new {
                success = true,
                message = message,
                originalEdgeKept = true,
                tangentEdgesFound = tangentEdgesArray.Length,
                tangentEdgesAdded = successCount,
                totalSelected = finalSelectedCount
            };
        }
        else
        {
            return new {
                success = false,
                error = "Found " + tangentEdgesArray.Length + " tangent edges but could not add any of them to the selection. Your original edge is still selected.",
                originalEdgeKept = true,
                tangentEdgesFound = tangentEdgesArray.Length,
                tangentEdgesAdded = 0,
                totalSelected = 1
            };
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Exception: " + ex.Message);
```

```
            Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
            return new { success = false, error = "Exception: " + ex.Message };
        }
    }
}
```

functi0nxy macro 2

Automation title : **I want a shortcut to fillet all remaining edges.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class FilletBoy
{
    public static object LetsFIllet()
    {
        try
        {
            Console.WriteLine("[START] Fillet all edges macro with user input dialog");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application"
};
            }

            Console.WriteLine("[INFO] Connected to SolidWorks");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a part." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Check if the active document is a part
            if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
            {
```

```
            return new { success = false, error = "Active document must be a part. Please
open a part document." };
        }

        Console.WriteLine("[INFO] Part document confirmed");

        // Show input dialog to get the fillet radius
        Console.WriteLine("[INFO] Showing input dialog for fillet radius");
        string radiusInput = ShowInputDialog("Enter fillet radius in mm:", "Fillet All Edges");

        if (string.IsNullOrEmpty(radiusInput))
        {
            Console.WriteLine("[INFO] User cancelled or provided empty input");
            return new { success = false, error = "No radius value provided", cancelled = true };
        }

        Console.WriteLine("[INFO] User provided radius: " + radiusInput);

        // Parse the radius value
        double filletRadius;
        if (!double.TryParse(radiusInput, out filletRadius))
        {
            return new { success = false, error = "Invalid radius value. Please enter a valid
number." };
        }

        // Convert from mm to meters (SolidWorks internal units)
        filletRadius = filletRadius / 1000.0;
        Console.WriteLine("[INFO] Fillet radius in meters: " + filletRadius);

        if (filletRadius <= 0)
        {
            return new { success = false, error = "Radius must be greater than 0." };
        }

        IFeatureManager swFeatMgr = swModel.FeatureManager as IFeatureManager;
        IPartDoc swPart = swModel as IPartDoc;

        // Get all edges from all bodies in the part
        Console.WriteLine("[INFO] Getting all edges from the part");
        System.Collections.Generic.List<IEdge> allEdges = GetAllEdges(swPart);

        if (allEdges.Count == 0)
        {
            return new { success = false, error = "No edges found in the part" };
        }

        Console.WriteLine("[INFO] Found " + allEdges.Count + " edges to fillet");
```

```
        // Clear any existing selection
        swModel.ClearSelection2(true);

        // Select all edges
        Console.WriteLine("[INFO] Selecting all edges for filleting");
        int selectedCount = 0;

        for (int i = 0; i < allEdges.Count; i++)
        {
            IEdge edge = allEdges[i];
            if (edge != null)
            {
                IEntity edgeEntity = edge as IEntity;
                if (edgeEntity != null)
                {
                    bool selectResult = edgeEntity.Select4(true, null);
                    if (selectResult)
                    {
                        selectedCount++;
                    }
                    else
                    {
                        Console.WriteLine("[WARNING] Failed to select edge " + i);
                    }
                }
            }
        }

        Console.WriteLine("[INFO] Successfully selected " + selectedCount + " edges out of "
+ allEdges.Count);

        if (selectedCount == 0)
        {
            return new { success = false, error = "Failed to select any edges for filleting" };
        }

        // Create the fillet feature
        Console.WriteLine("[INFO] Creating fillet feature with radius " + (filletRadius * 1000) +
" mm");
        bool filletSuccess = CreateConstantRadiusFillet(swFeatMgr, filletRadius, "All Edges
Fillet (" + (filletRadius * 1000) + "mm)");

        // Clear selection
        swModel.ClearSelection2(true);

        // Rebuild the model
        Console.WriteLine("[INFO] Rebuilding model");
```

```
            swModel.ForceRebuild3(false);

            if (filletSuccess)
            {
                return new {
                    success = true,
                    message = "✅ Successfully created fillet on " + selectedCount + " edges with
radius " + (filletRadius * 1000) + " mm",
                    edgesProcessed = selectedCount,
                    radiusMm = filletRadius * 1000
                };
            }
            else
            {
                return new {
                    success = false,
                    error = "Failed to create fillet feature. This could be due to geometric constraints
or edges unsuitable for the specified radius."
                };
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("[ERROR] Main exception: " + ex.Message);
            return new { success = false, error = "Exception: " + ex.Message };
        }
    }

    private static System.Collections.Generic.List<IEdge> GetAllEdges(IPartDoc swPart)
    {
        System.Collections.Generic.List<IEdge> allEdges = new
System.Collections.Generic.List<IEdge>();

        try
        {
            Console.WriteLine("[DEBUG] Getting all edges from part bodies");

            // Get all solid bodies
            object[] bodies = swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as
object[];

            if (bodies == null || bodies.Length == 0)
            {
                Console.WriteLine("[WARNING] No solid bodies found in the part");
                return allEdges;
            }

            Console.WriteLine("[DEBUG] Found " + bodies.Length + " solid bodies");
```

```
            foreach (object bodyObj in bodies)
            {
                IBody2 swBody = bodyObj as IBody2;
                if (swBody == null) continue;

                Console.WriteLine("[DEBUG] Processing body: " + swBody.Name);

                // Get all edges from the body
                object[] edges = swBody.GetEdges() as object[];
                if (edges == null) continue;

                Console.WriteLine("[DEBUG] Found " + edges.Length + " edges in body");

                for (int i = 0; i < edges.Length; i++)
                {
                    IEdge swEdge = edges[i] as IEdge;
                    if (swEdge != null)
                    {
                        allEdges.Add(swEdge);
                    }
                }
            }

            Console.WriteLine("[DEBUG] Total edges collected: " + allEdges.Count);
            return allEdges;
        }
        catch (Exception ex)
        {
            Console.WriteLine("[ERROR] Error getting all edges: " + ex.Message);
            return allEdges;
        }
    }

    private static bool CreateConstantRadiusFillet(IFeatureManager swFeatMgr, double
radius, string filletName)
    {
        try
        {
            Console.WriteLine("[DEBUG] Creating constant radius fillet with radius: " + radius + "
meters (" + (radius * 1000) + " mm)");

            // Create the fillet feature data object using modern API
            object swFeatDataObj =
swFeatMgr.CreateDefinition((int)swFeatureNameID_e.swFmFillet);
            ISimpleFilletFeatureData2 swFilletData = swFeatDataObj as
ISimpleFilletFeatureData2;
```

```
        if (swFilletData == null)
        {
            Console.WriteLine("[ERROR] Failed to create fillet feature data object");
            return false;
        }

        Console.WriteLine("[DEBUG] Successfully created fillet feature data object");

        // Initialize the fillet feature data to constant radius fillet
        bool initResult =
swFilletData.Initialize((int)swSimpleFilletType_e.swConstRadiusFillet);
        if (!initResult)
        {
            Console.WriteLine("[ERROR] Failed to initialize fillet data");
            return false;
        }

        Console.WriteLine("[DEBUG] Successfully initialized fillet data as constant radius
fillet");

        // Set the cross-section profile to circular (standard fillet)
        swFilletData.ConicTypeForCrossSectionProfile =
(int)swFeatureFilletProfileType_e.swFeatureFilletCircular;

        // Set the default radius
        swFilletData.DefaultRadius = radius;

        // Set overflow type to default
        swFilletData.OverflowType =
(int)swFilletOverFlowType_e.swFilletOverFlowType_Default;

        // Optional settings
        swFilletData.KeepFeatures = true;
        swFilletData.PropagateToTangentFaces = false;
        swFilletData.RoundCorners = true;

        Console.WriteLine("[DEBUG] Set all fillet properties, now creating feature");

        // Create the fillet feature
        IFeature swFeat = swFeatMgr.CreateFeature(swFilletData) as IFeature;

        if (swFeat != null)
        {
            Console.WriteLine("[SUCCESS] Created fillet feature: " + swFeat.Name);

            // Optionally rename the feature
            if (!string.IsNullOrEmpty(filletName))
            {
```

```
                swFeat.Name = filletName;
                Console.WriteLine("[DEBUG] Renamed feature to: " + filletName);
            }

            return true;
        }
        else
        {
            Console.WriteLine("[ERROR] Failed to create fillet feature - CreateFeature
returned null");
            return false;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Exception in CreateConstantRadiusFillet: " +
ex.Message);
        return false;
    }
}

private static string ShowInputDialog(string prompt, string title)
{
    try
    {
        Console.WriteLine("[DEBUG] Creating input dialog");

        // Create a simple input form
        System.Windows.Forms.Form inputForm = new System.Windows.Forms.Form();
        inputForm.Text = title;
        inputForm.Width = 400;
        inputForm.Height = 160;
        inputForm.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
        inputForm.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedDialog;
        inputForm.MaximizeBox = false;
        inputForm.MinimizeBox = false;

        // Add label
        System.Windows.Forms.Label label = new System.Windows.Forms.Label();
        label.Text = prompt;
        label.Left = 10;
        label.Top = 20;
        label.Width = 360;
        label.Height = 20;

        // Add text box
        System.Windows.Forms.TextBox textBox = new System.Windows.Forms.TextBox();
```

```
        textBox.Left = 10;
        textBox.Top = 45;
        textBox.Width = 280;
        textBox.Text = "2"; // Default value of 2mm
        textBox.SelectAll(); // Select all text for easy replacement

        // Add OK button
        System.Windows.Forms.Button okButton = new System.Windows.Forms.Button();
        okButton.Text = "OK";
        okButton.Left = 210;
        okButton.Top = 80;
        okButton.Width = 75;
        okButton.DialogResult = System.Windows.Forms.DialogResult.OK;

        // Add Cancel button
        System.Windows.Forms.Button cancelButton = new
System.Windows.Forms.Button();
        cancelButton.Text = "Cancel";
        cancelButton.Left = 295;
        cancelButton.Top = 80;
        cancelButton.Width = 75;
        cancelButton.DialogResult = System.Windows.Forms.DialogResult.Cancel;

        // Add controls to form
        inputForm.Controls.Add(label);
        inputForm.Controls.Add(textBox);
        inputForm.Controls.Add(okButton);
        inputForm.Controls.Add(cancelButton);

        // Set accept and cancel buttons
        inputForm.AcceptButton = okButton;
        inputForm.CancelButton = cancelButton;

        // Focus on textbox
        textBox.Focus();

        Console.WriteLine("[DEBUG] Showing input dialog to user");

        // Show dialog and get result
        System.Windows.Forms.DialogResult result = inputForm.ShowDialog();

        if (result == System.Windows.Forms.DialogResult.OK)
        {
            Console.WriteLine("[DEBUG] User clicked OK, returning input: " + textBox.Text);
            return textBox.Text.Trim();
        }
        else
        {
```

```
            Console.WriteLine("[DEBUG] User cancelled input dialog");
            return string.Empty;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Input dialog error: " + ex.Message);
        return string.Empty;
    }
  }
}
```

functi0nxy macro 3

Automation title : **I want a macro to resize all dimensions by a factor.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class Resizer
{
    public static object Resize()
    {
        try
        {
            Console.WriteLine("[START] Beginning sketch dimensions scaling macro - FINAL
FIX");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks
application" };
            }

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
                return new { success = false, error = "No document is open. Please open a part
document." };
            }

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Show scale factor input dialog
            Console.WriteLine("[INFO] Showing scale factor input dialog");
            string scaleFactorInput = ShowScaleFactorDialog();
```

```
            if (string.IsNullOrEmpty(scaleFactorInput))
            {
                Console.WriteLine("[INFO] User cancelled scale factor input");
                return new { success = false, error = "Scale factor input cancelled by user",
cancelled = true };
            }

            // Parse scale factor
            double scaleFactor;
            if (!double.TryParse(scaleFactorInput, out scaleFactor))
            {
                return new { success = false, error = "Invalid scale factor. Please enter a valid
number." };
            }

            if (scaleFactor <= 0)
            {
                return new { success = false, error = "Scale factor must be greater than 0." };
            }

            Console.WriteLine("[INFO] Scale factor entered: " + scaleFactor);

            // Get current active configuration name
            string activeConfigName =
swModel.ConfigurationManager.ActiveConfiguration.Name;
            Console.WriteLine("[INFO] Active configuration: " + activeConfigName);

            // Start traversing features for dimensions
            Console.WriteLine("[INFO] Starting to traverse all features for dimensions");

            int sketchCount = 0;
            int dimensionsScaled = 0;

            // Get the first feature to start traversal - VBA pattern
            IFeature swFeat = swModel.FirstFeature() as IFeature;

            // Process each feature to find dimensions - following VBA traverse pattern
            while (swFeat != null)
            {
                try
                {
                    string featureType = swFeat.GetTypeName2();
                    Console.WriteLine("[DEBUG] Processing feature: " + swFeat.Name + " (Type:
" + featureType + ")");

                    // Check if this is a sketch feature (ProfileFeature) - like VBA example
                    if (featureType == "ProfileFeature")
                    {
```

```
                    Console.WriteLine("[INFO] Found sketch: " + swFeat.Name);
                    sketchCount++;

                    // Get display dimensions from feature - VBA pattern
                    IDisplayDimension swDispDim = swFeat.GetFirstDisplayDimension() as
IDisplayDimension;

                    int dimCount = 0;

                    // Process all display dimensions in this feature - VBA pattern
                    while (swDispDim != null)
                    {
                       try
                       {
                          dimCount++;
                          Console.WriteLine("[DEBUG] Processing display dimension " +
dimCount);

                          // Get the underlying dimension using VBA pattern
                          IDimension swDim = swDispDim.GetDimension2(0) as IDimension;

                          if (swDim != null)
                          {
                             Console.WriteLine("[DEBUG] Got dimension: " +
swDim.GetNameForSelection());

                             double currentValue = 0.0;
                             bool valueObtained = false;

                             // Try multiple methods to get dimension value - FIXED APPROACH
                             try
                             {
                                // Method 1: Try IGetSystemValue3 which returns double directly
                                currentValue =
swDim.IGetSystemValue3((int)swInConfigurationOpts_e.swThisConfiguration, 0, ref
activeConfigName);
                                valueObtained = true;
                                Console.WriteLine("[DEBUG] Got value using IGetSystemValue3:
" + currentValue);
                             }
                             catch (System.Exception ex1)
                             {
                                Console.WriteLine("[DEBUG] IGetSystemValue3 failed: " +
ex1.Message);

                                // Method 2: Try GetSystemValue3 with null config names
                                try
                                {
```

```
                                object valObj =
swDim.GetSystemValue3((int)swInConfigurationOpts_e.swThisConfiguration, null);

                                if (valObj != null)
                                {
                                    if (valObj is System.Array)
                                    {
                                        object[] valArray = valObj as object[];
                                        if (valArray != null && valArray.Length > 0)
                                        {
                                            currentValue = (double)valArray[0];
                                            valueObtained = true;
                                            Console.WriteLine("[DEBUG] Got value from array[0]: "
+ currentValue);
                                        }
                                    }
                                    else
                                    {
                                        currentValue = (double)valObj;
                                        valueObtained = true;
                                        Console.WriteLine("[DEBUG] Got direct value: " +
currentValue);
                                    }
                                }
                            }
                            catch (System.Exception ex2)
                            {
                                Console.WriteLine("[DEBUG] GetSystemValue3 with null
failed: " + ex2.Message);

                                // Method 3: Try with specific configuration
                                try
                                {
                                    object valObj2 =
swDim.GetSystemValue3((int)swInConfigurationOpts_e.swSpecifyConfiguration, new string[]
{ activeConfigName });

                                    if (valObj2 != null)
                                    {
                                        if (valObj2 is System.Array)
                                        {
                                            object[] valArray2 = valObj2 as object[];
                                            if (valArray2 != null && valArray2.Length > 0)
                                            {
                                                currentValue = (double)valArray2[0];
                                                valueObtained = true;
                                                Console.WriteLine("[DEBUG] Got value from specify
config array[0]: " + currentValue);
```

```
                    }
                }
                else
                {
                    currentValue = (double)valObj2;
                    valueObtained = true;
                    Console.WriteLine("[DEBUG] Got direct value from
specify config: " + currentValue);
                }
            }
        }
        catch (System.Exception ex3)
        {
            Console.WriteLine("[WARNING] All methods failed for
dimension: " + ex3.Message);
        }
    }
}

if (valueObtained && currentValue > 0)
{
    double newValue = currentValue * scaleFactor;

    Console.WriteLine("[DEBUG] Scaling " +
swDim.GetNameForSelection() + " from " + (currentValue * 1000).ToString("F1") + "mm to "
+ (newValue * 1000).ToString("F1") + "mm");

    // Set new dimension value - try multiple methods
    int setResult = -1;
    try
    {
        setResult = swDim.SetSystemValue3(newValue,
(int)swInConfigurationOpts_e.swThisConfiguration, null);
    }
    catch
    {
        try
        {
            setResult = swDim.SetSystemValue3(newValue,
(int)swInConfigurationOpts_e.swSpecifyConfiguration, new string[] { activeConfigName });
        }
        catch (System.Exception setEx)
        {
            Console.WriteLine("[WARNING] Failed to set dimension
value: " + setEx.Message);
        }
    }
```

```
                                if (setResult ==
(int)swSetValueReturnStatus_e.swSetValue_Successful)
                                {
                                    dimensionsScaled++;
                                    Console.WriteLine("[SUCCESS] Dimension " +
swDim.GetNameForSelection() + " scaled successfully");
                                }
                                else
                                {
                                    Console.WriteLine("[WARNING] Failed to set dimension " +
swDim.GetNameForSelection() + ", code: " + setResult);
                                }
                            }
                            else
                            {
                                Console.WriteLine("[WARNING] Could not obtain value for
dimension " + swDim.GetNameForSelection());
                            }
                        }
                        else
                        {
                            Console.WriteLine("[WARNING] Could not get IDimension from
IDisplayDimension");
                        }

                        // Get next display dimension using VBA pattern
                        swDispDim = swFeat.GetNextDisplayDimension(swDispDim) as
IDisplayDimension;
                    }
                    catch (System.Exception dimEx)
                    {
                        Console.WriteLine("[WARNING] Error with dimension: " +
dimEx.Message);
                        break;
                    }
                }

                Console.WriteLine("[INFO] Processed " + dimCount + " dimensions in
sketch: " + swFeat.Name);
            }

            // Get next feature - VBA pattern
            swFeat = swFeat.GetNextFeature() as IFeature;
        }
        catch (System.Exception featEx)
        {
            Console.WriteLine("[WARNING] Error with feature: " + featEx.Message);
            // Continue with next feature
```

```
                try
                {
                    swFeat = swFeat.GetNextFeature() as IFeature;
                }
                catch
                {
                    Console.WriteLine("[WARNING] Cannot get next feature, stopping");
                    break;
                }
            }
        }

        Console.WriteLine("[INFO] Processed " + sketchCount + " sketches");

        if (sketchCount == 0)
        {
            return new { success = false, error = "No sketches found in the document" };
        }

        if (dimensionsScaled == 0)
        {
            return new { success = false, error = "No dimensions found or scaled in
sketches. The dimensions might be driven by other features or constraints." };
        }

        // Rebuild the model to update all changes
        Console.WriteLine("[INFO] Rebuilding model...");
        swModel.ForceRebuild3(false);

        Console.WriteLine("[SUCCESS] Sketch dimensions scaling completed!");

        return new {
            success = true,
            message = "Successfully scaled " + dimensionsScaled + " dimensions in " +
sketchCount + " sketches by factor " + scaleFactor,
            sketchesProcessed = sketchCount,
            dimensionsScaled = dimensionsScaled,
            scaleFactor = scaleFactor
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Exception: " + ex.Message);
        return new { success = false, error = "Exception: " + ex.Message };
    }
}

private static string ShowScaleFactorDialog()
```

```
    {
        try
        {
            Console.WriteLine("[DEBUG] Creating scale factor input dialog");

            // Create input form
            System.Windows.Forms.Form inputForm = new System.Windows.Forms.Form();
            inputForm.Text = "Scale Sketch Dimensions";
            inputForm.Width = 450;
            inputForm.Height = 180;
            inputForm.StartPosition =
System.Windows.Forms.FormStartPosition.CenterScreen;
            inputForm.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedDialog;
            inputForm.MaximizeBox = false;
            inputForm.MinimizeBox = false;

            // Add instructions label
            System.Windows.Forms.Label instructionLabel = new
System.Windows.Forms.Label();
            instructionLabel.Text = "Enter scale factor to multiply all sketch dimensions:";
            instructionLabel.Left = 10;
            instructionLabel.Top = 15;
            instructionLabel.Width = 400;
            instructionLabel.Height = 20;

            // Add example label
            System.Windows.Forms.Label exampleLabel = new
System.Windows.Forms.Label();
            exampleLabel.Text = "Examples: 2.0 (double size), 0.5 (half size), 1.5 (150% size)";
            exampleLabel.Left = 10;
            exampleLabel.Top = 40;
            exampleLabel.Width = 400;
            exampleLabel.Height = 20;

            // Add scale factor label
            System.Windows.Forms.Label scaleLabel = new System.Windows.Forms.Label();
            scaleLabel.Text = "Scale Factor:";
            scaleLabel.Left = 10;
            scaleLabel.Top = 70;
            scaleLabel.Width = 100;
            scaleLabel.Height = 23;

            // Add text box for scale factor
            System.Windows.Forms.TextBox scaleTextBox = new
System.Windows.Forms.TextBox();
            scaleTextBox.Left = 120;
            scaleTextBox.Top = 70;
```

```
            scaleTextBox.Width = 150;
            scaleTextBox.Text = "1.0";
            scaleTextBox.SelectAll(); // Select all text for easy replacement

            // Add OK button
            System.Windows.Forms.Button okButton = new System.Windows.Forms.Button();
            okButton.Text = "Scale Dimensions";
            okButton.Left = 280;
            okButton.Top = 68;
            okButton.Width = 120;
            okButton.Height = 27;
            okButton.DialogResult = System.Windows.Forms.DialogResult.OK;

            // Add Cancel button
            System.Windows.Forms.Button cancelButton = new
System.Windows.Forms.Button();
            cancelButton.Text = "Cancel";
            cancelButton.Left = 280;
            cancelButton.Top = 100;
            cancelButton.Width = 120;
            cancelButton.Height = 27;
            cancelButton.DialogResult = System.Windows.Forms.DialogResult.Cancel;

            // Add controls to form
            inputForm.Controls.Add(instructionLabel);
            inputForm.Controls.Add(exampleLabel);
            inputForm.Controls.Add(scaleLabel);
            inputForm.Controls.Add(scaleTextBox);
            inputForm.Controls.Add(okButton);
            inputForm.Controls.Add(cancelButton);

            // Set accept and cancel buttons
            inputForm.AcceptButton = okButton;
            inputForm.CancelButton = cancelButton;

            // Set focus to text box
            scaleTextBox.Focus();

            Console.WriteLine("[DEBUG] Showing scale factor dialog to user");

            // Show dialog and get result
            System.Windows.Forms.DialogResult result = inputForm.ShowDialog();

            if (result == System.Windows.Forms.DialogResult.OK)
            {
                string scaleInput = scaleTextBox.Text.Trim();
                Console.WriteLine("[DEBUG] User entered scale factor: " + scaleInput);
                return scaleInput;
```

```
            }
            else
            {
                Console.WriteLine("[DEBUG] User cancelled scale factor dialog");
                return string.Empty;
            }
        }
        catch (System.Exception ex)
        {
            Console.WriteLine("[ERROR] Scale factor dialog error: " + ex.Message);
            return string.Empty;
        }
    }
}
```

ThaGuvnor

Automation title : A macro that would work in a part or assembly file that hides all planes.

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public static object HidePlanes()
{
    try
    {
        Console.WriteLine("[START] Beginning toggle all reference planes visibility macro");

        // Create SolidWorks instance (will be fixed by executor to use Marshal.GetActiveObject)
        SldWorks swApp = Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

        if (swApp == null)
        {
            return new { success = false, error = "Could not connect to SolidWorks application" };
        }

        Console.WriteLine("[INFO] Connected to SolidWorks");

        // Get the active document
        IModelDoc2 swModel = swApp.IActiveDoc2;

        if (swModel == null)
```

```
        {
            return new { success = false, error = "No document is open. Please open a document." };
        }

        Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

        // Check current visibility state of reference planes
        bool currentlyVisible = swModel.GetVisibilityOfConstructPlanes();
        Console.WriteLine("[INFO] Reference planes currently visible: " + currentlyVisible);

        // Toggle the visibility of all reference planes
        Console.WriteLine("[INFO] Toggling reference planes visibility...");
        swModel.ViewDispRefplanes();

        // Check new state after toggle
        bool newVisibilityState = swModel.GetVisibilityOfConstructPlanes();
        Console.WriteLine("[INFO] Reference planes now visible: " + newVisibilityState);

        // Refresh the graphics to ensure changes are displayed
        Console.WriteLine("[INFO] Refreshing graphics...");
        swModel.GraphicsRedraw2();

        string action = newVisibilityState ? "shown" : "hidden";
        string message = "Successfully " + action + " all reference planes in the graphics area";

        Console.WriteLine("[SUCCESS] " + message);
```

```
            return new {
                success = true,
                message = message,
                previousState = currentlyVisible,
                newState = newVisibilityState,
                action = action
            };
        }
        catch (Exception ex)
        {
            Console.WriteLine("[ERROR] Exception occurred: " + ex.Message);
            return new { success = false, error = "Exception: " + ex.Message };
        }
    }
}
```

# Zombie_Joe_Knives

Automation title : **A macro to automatically save a .stl file in the same location that the part file is located in.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
public static object save()
{
  try
  {
    Console.WriteLine("[START] Beginning automatic STL export macro");

    // Create SolidWorks instance (will be fixed by executor to use Marshal.GetActiveObject)
    SldWorks swApp = Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

    if (swApp == null)
    {
      return new { success = false, error = "Could not connect to SolidWorks application" };
    }

    Console.WriteLine("[INFO] Connected to SolidWorks successfully");

    // Get the active document
    IModelDoc2 swModel = swApp.IActiveDoc2;

    if (swModel == null)
    {
      return new { success = false, error = "No document is open. Please open a part document." };
    }
```

```
        Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

        // Check if the active document is a part
        if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
        {
            return new { success = false, error = "Active document is not a part. This macro works only with part documents. Please open a part file." };
        }

        // Get the current file path
        string currentPath = swModel.GetPathName();
        Console.WriteLine("[INFO] Current document path: " + currentPath);

        if (string.IsNullOrEmpty(currentPath))
        {
            return new { success = false, error = "The current document has not been saved yet. Please save the part file first before exporting to STL." };
        }

        // Verify the file exists
        if (!System.IO.File.Exists(currentPath))
        {
            return new { success = false, error = "The current document path does not exist: " + currentPath };
        }

        // Create STL file path in the same directory
        string directory = System.IO.Path.GetDirectoryName(currentPath);
        string fileNameWithoutExt = System.IO.Path.GetFileNameWithoutExtension(currentPath);
        string stlFilePath = System.IO.Path.Combine(directory, fileNameWithoutExt + ".stl");

        Console.WriteLine("[INFO] Will export STL to: " + stlFilePath);
```

```
        // Clear any selections to ensure the entire model is exported
        swModel.ClearSelection2(true);
        Console.WriteLine("[DEBUG] Cleared all selections");

        // Export as STL using SaveAs3
        Console.WriteLine("[DEBUG] Starting STL export...");
        int saveErrors = 0;
        int saveWarnings = 0;

        bool saveResult = swModel.Extension.SaveAs(
            stlFilePath,                                // File path
            (int)swSaveAsVersion_e.swSaveAsCurrentVersion,       // Version
            (int)swSaveAsOptions_e.swSaveAsOptions_Silent,       // Options - silent mode
            null,                                       // Export data (not used for STL)
            ref saveErrors,                             // Errors
            ref saveWarnings                            // Warnings
        );

        if (!saveResult)
        {
            Console.WriteLine("[ERROR] Failed to save STL file. Errors: " + saveErrors + ", Warnings: " +
saveWarnings);

            string errorMsg = "Failed to export STL file.";
            if (saveErrors != 0)
            {
                errorMsg += " Error code: " + saveErrors;
            }

            return new { success = false, error = errorMsg };
```

```
        }

        // Verify the STL file was created
        if (!System.IO.File.Exists(stlFilePath))
        {
            Console.WriteLine("[ERROR] STL file was not created at expected location: " + stlFilePath);
            return new { success = false, error = "STL file was not created successfully at: " + stlFilePath };
        }

        Console.WriteLine("[SUCCESS] STL file exported successfully");

        // Get file info for the response
        System.IO.FileInfo stlFileInfo = new System.IO.FileInfo(stlFilePath);
        Console.WriteLine("[INFO] STL file size: " + (stlFileInfo.Length / 1024) + " KB");

        Console.WriteLine("[COMPLETE] Macro execution finished successfully");

        return new {
            success = true,
            message = "Successfully exported STL file '" + System.IO.Path.GetFileName(stlFilePath) + "' to the same
directory as the part file",
            partFile = currentPath,
            stlFile = stlFilePath,
            stlFileName = System.IO.Path.GetFileName(stlFilePath),
            directory = directory,
            fileSizeKB = stlFileInfo.Length / 1024
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Exception occurred: " + ex.Message);
```

```
        Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
        return new { success = false, error = "Exception: " + ex.Message };
    }
}
```

FightPillow

Automation title : *macro that adds the radius in the feature tree for the filets.*

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public static object Rename()
{
    try
    {
        Console.WriteLine("[START] Beginning fillet feature rename macro");

        // Create SolidWorks instance (will be fixed by executor to use Marshal.GetActiveObject)
        SldWorks swApp = Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

        if (swApp == null)
        {
            return new { success = false, error = "Could not connect to SolidWorks application" };
        }

        Console.WriteLine("[INFO] Connected to SolidWorks");

        // Get the active document
        IModelDoc2 swModel = swApp.IActiveDoc2;

        if (swModel == null)
        {
            return new { success = false, error = "No document is open. Please open a part or assembly document." };
        }

        Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

        // Track renamed fillets
        int renamedFillets = 0;
        int totalFillets = 0;

        // Disable feature tree for performance
        IFeatureManager swFeatMgr = swModel.FeatureManager;
        if (swFeatMgr != null)
        {
            swFeatMgr.EnableFeatureTree = false;
            swFeatMgr.EnableFeatureTreeWindow = false;
        }

        Console.WriteLine("[INFO] Searching for fillet features...");

        // Iterate through all features
        IFeature swFeat = swModel.IFirstFeature();
        while (swFeat != null)
        {
            ProcessFeature(swFeat, swModel, ref renamedFillets, ref totalFillets);
            swFeat = swFeat.IGetNextFeature();
        }

        Console.WriteLine("[COMPLETE] Processed all features");

        // Re-enable feature tree
        if (swFeatMgr != null)
        {
            swFeatMgr.EnableFeatureTree = true;
            swFeatMgr.EnableFeatureTreeWindow = true;
        }

        if (totalFillets == 0)
        {
            return new {
                success = false,
                error = "No fillet features found in the current document."
            };
        }

        if (renamedFillets > 0)
        {
            // Rebuild the model to refresh the feature tree display
```

```
            Console.WriteLine("[INFO] Rebuilding model to refresh display...");
            swModel.ForceRebuild3(false);

            return new {
              success = true,
              message = "Successfully renamed " + renamedFillets + " out of " + totalFillets + " fillet feature(s) with radius information",
              renamedCount = renamedFillets,
              totalFilletCount = totalFillets
            };
          }
          else
          {
            return new {
              success = true,
              message = "Found " + totalFillets + " fillet feature(s) but none needed renaming (already had radius information or couldn't access radius
data)",
              renamedCount = 0,
              totalFilletCount = totalFillets
            };
          }
        }
        catch (System.Exception ex)
        {
          Console.WriteLine("[ERROR] Main exception: " + ex.Message);
          return new { success = false, error = "Exception: " + ex.Message };
        }
      }

      private static void ProcessFeature(IFeature swFeat, IModelDoc2 swModel, ref int renamedFillets, ref int totalFillets)
      {
        try
        {
          string featTypeName = swFeat.GetTypeName2();

          if (featTypeName == "Fillet")
          {
            totalFillets++;
            Console.WriteLine("[INFO] Found fillet feature: " + swFeat.Name + " (Type: " + featTypeName + ")");

            if (RenameFilletWithRadius(swFeat, swModel))
            {
              renamedFillets++;
            }
          }

          // Process sub-features recursively
          IFeature swSubFeat = swFeat.IGetFirstSubFeature();
          while (swSubFeat != null)
          {
            ProcessFeature(swSubFeat, swModel, ref renamedFillets, ref totalFillets);
            swSubFeat = swSubFeat.IGetNextSubFeature();
          }
        }
        catch (System.Exception ex)
        {
          Console.WriteLine("[WARNING] Error processing feature " + (swFeat != null ? swFeat.Name : "null") + ": " + ex.Message);
        }
      }

      private static bool RenameFilletWithRadius(IFeature swFeat, IModelDoc2 swModel)
      {
        try
        {
          Console.WriteLine("[DEBUG] Processing fillet: " + swFeat.Name);

          // Get the feature definition
          object swFeatDefObj = swFeat.GetDefinition();

          if (swFeatDefObj == null)
          {
            Console.WriteLine("[WARNING] Could not get definition for fillet: " + swFeat.Name);
            return false;
          }

          string radiusInfo = "";

          // Try as simple fillet first
          ISimpleFilletFeatureData2 swSimpleFilletData = swFeatDefObj as ISimpleFilletFeatureData2;
          if (swSimpleFilletData != null)
          {
            Console.WriteLine("[DEBUG] Processing as simple fillet feature");

            // Access selections to read the feature data
            if (swSimpleFilletData.AccessSelections(swModel, null))
            {
              try
```

```
    {
      // Get the default radius
      double defaultRadius = swSimpleFilletData.DefaultRadius;
      Console.WriteLine("[DEBUG] Default radius: " + (defaultRadius * 1000) + " mm");

      if (swSimpleFilletData.IsMultipleRadius)
      {
        Console.WriteLine("[DEBUG] Feature has multiple radii");
        int filletItemsCount = swSimpleFilletData.FilletItemsCount;

        if (filletItemsCount > 1)
        {
          // Multiple radii - show range or first few
          List<double> radii = new List<double>();
          for (int i = 0; i < Math.Min(filletItemsCount, 3); i++) // Limit to first 3 radii
          {
            double radius = swSimpleFilletData.GetRadius(i);
            radii.Add(radius * 1000); // Convert to mm
          }

          if (radii.Count == 1)
          {
            radiusInfo = " R" + radii[0].ToString("0.##");
          }
          else if (radii.Count == 2 && Math.Abs(radii[0] - radii[1]) < 0.001)
          {
            // Same radius
            radiusInfo = " R" + radii[0].ToString("0.##");
          }
          else
          {
            // Different radii
            radiusInfo = " R" + radii[0].ToString("0.##") + "-" + radii[radii.Count - 1].ToString("0.##");
          }
        }
        else
        {
          radiusInfo = " R" + (defaultRadius * 1000).ToString("0.##");
        }
      }
      else
      {
        // Single radius
        radiusInfo = " R" + (defaultRadius * 1000).ToString("0.##");
      }
    }
    finally
    {
      swSimpleFilletData.ReleaseSelectionAccess();
    }
  }
  else
  {
    Console.WriteLine("[WARNING] Failed to access selections for simple fillet: " + swFeat.Name);
    return false;
  }
}
else
{
  // Try as variable fillet
  IVariableFilletFeatureData2 swVarFilletData = swFeatDefObj as IVariableFilletFeatureData2;
  if (swVarFilletData != null)
  {
    Console.WriteLine("[DEBUG] Processing as variable fillet feature");

    // Access selections to read the feature data
    if (swVarFilletData.AccessSelections(swModel, null))
    {
      try
      {
        // For variable fillets, use default radius or indicate as variable
        double defaultRadius = swVarFilletData.DefaultRadius;
        radiusInfo = " R" + (defaultRadius * 1000).ToString("0.##") + "var";
      }
      finally
      {
        swVarFilletData.ReleaseSelectionAccess();
      }
    }
    else
    {
      Console.WriteLine("[WARNING] Failed to access selections for variable fillet: " + swFeat.Name);
      return false;
    }
  }
  else
```

```
            {
                Console.WriteLine("[WARNING] Could not cast fillet to known data type: " + swFeat.Name);
                return false;
            }
        }

        if (!string.IsNullOrEmpty(radiusInfo))
        {
            // Check if the feature name already contains radius information
            string currentName = swFeat.Name;

            // Simple check - if it already contains " R" followed by numbers, skip it
            if (currentName.Contains(" R") && System.Text.RegularExpressions.Regex.IsMatch(currentName, @" R\d+"))
            {
                Console.WriteLine("[INFO] Feature " + currentName + " already contains radius information, skipping");
                return false;
            }

            string newName = currentName + radiusInfo;

            // Check if the new name would conflict with existing features
            string finalName = newName;
            int counter = 1;
            IFeatureManager swFeatMgr = swModel.FeatureManager;

            // Fixed: Cast enum to int
            while (swFeatMgr.IsNameUsed((int)swNameType_e.swFeatureName, finalName))
            {
                finalName = newName + "_" + counter;
                counter++;
                if (counter > 100) // Safety check
                {
                    Console.WriteLine("[WARNING] Could not find unique name for " + newName);
                    return false;
                }
            }

            Console.WriteLine("[SUCCESS] Renaming '" + currentName + "' to '" + finalName + "'");
            swFeat.Name = finalName;

            return true;
        }
        else
        {
            Console.WriteLine("[WARNING] Could not determine radius for fillet: " + swFeat.Name);
            return false;
        }
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Exception while processing fillet " + swFeat.Name + ": " + ex.Message);
        return false;
    }
}
```

Staffchild101

Automation title : Loop through a part or assembly and save out all of the bodies as either STEP or STL, choose file type, add suffix date in YYMMDD.

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

GB5897

Automation title : **Sheet metal flatten and export face to dxf.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class SheetMetalExport
{
    // Sheet metal export options (converted from VBA enum)
    private enum SheetMetalOptions_e
    {
        ExportFlatPatternGeometry = 1,
        IncludeHiddenEdges = 2,
        ExportBendLines = 4,
        IncludeSketches = 8,
        MergeCoplanarFaces = 16,
        ExportLibraryFeatures = 32,
        ExportFormingTools = 64,
        ExportBoundingBox = 2048
    }

    public static object StartTheExport()
    {
        try
        {
            Console.WriteLine("[START] Beginning bulk sheet metal DXF export from
assembly");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;

            if (swApp == null)
            {
                return new { success = false, error = "Could not connect to SolidWorks application"
};
            }

            Console.WriteLine("[INFO] Connected to SolidWorks");

            // Get the active document
```

```
        IModelDoc2 swModel = swApp.IActiveDoc2;

        if (swModel == null)
        {
            return new { success = false, error = "No document is open. Please open an
assembly." };
        }

        Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

        // Check if the active document is an assembly
        if (swModel.GetType() != (int)swDocumentTypes_e.swDocASSEMBLY)
        {
            return new { success = false, error = "Active document is not an assembly. Please
open an assembly document." };
        }

        IAssemblyDoc swAssy = swModel as IAssemblyDoc;
        if (swAssy == null)
        {
            return new { success = false, error = "Failed to cast to assembly document" };
        }

        Console.WriteLine("[INFO] Assembly document confirmed");

        // Get all components from the assembly
        object[] vComps = swAssy.GetComponents(false) as object[];

        if (vComps == null || vComps.Length == 0)
        {
            return new { success = false, error = "No components found in the assembly" };
        }

        Console.WriteLine("[INFO] Found " + vComps.Length + " components in assembly");

        // Get assembly directory for output path
        string assemblyPath = swModel.GetPathName();
        if (string.IsNullOrEmpty(assemblyPath))
        {
            return new { success = false, error = "Assembly must be saved before exporting
DXF files. Please save the assembly first." };
        }

        string assemblyDir = System.IO.Path.GetDirectoryName(assemblyPath);
        string dxfOutputDir = System.IO.Path.Combine(assemblyDir, "DXF_Exports");

        // Create DXF output directory if it doesn't exist
        if (!System.IO.Directory.Exists(dxfOutputDir))
```

```
        {
            System.IO.Directory.CreateDirectory(dxfOutputDir);
            Console.WriteLine("[INFO] Created DXF output directory: " + dxfOutputDir);
        }

        int exportedCount = 0;
        int skippedCount = 0;
        System.Collections.Generic.List<string> processedFiles = new
System.Collections.Generic.List<string>();
        System.Collections.Generic.List<string> exportedFiles = new
System.Collections.Generic.List<string>();

        // Process each component
        for (int i = 0; i < vComps.Length; i++)
        {
            IComponent2 swComp = vComps[i] as IComponent2;

            if (swComp == null)
            {
                Console.WriteLine("[WARN] Component " + i + " is null, skipping");
                skippedCount++;
                continue;
            }

            // Skip suppressed components
            if (swComp.GetSuppression() ==
(int)swComponentSuppressionState_e.swComponentSuppressed)
            {
                Console.WriteLine("[INFO] Skipping suppressed component: " +
swComp.Name2);
                skippedCount++;
                continue;
            }

            // Only process part components
            string compPath = swComp.GetPathName();
            if (string.IsNullOrEmpty(compPath) || !compPath.ToLower().EndsWith(".sldprt"))
            {
                Console.WriteLine("[INFO] Skipping non-part component: " + swComp.Name2);
                skippedCount++;
                continue;
            }

            // Skip if already processed (avoid duplicates)
            if (processedFiles.Contains(compPath))
            {
                Console.WriteLine("[INFO] Skipping duplicate part: " +
System.IO.Path.GetFileName(compPath));
```

```
                skippedCount++;
                continue;
            }

            processedFiles.Add(compPath);
            Console.WriteLine("[INFO] Processing part component: " + swComp.Name2 + " ("
+ (i + 1) + "/" + vComps.Length + ")");

            try
            {
                // Open the component in its own window
                IDocumentSpecification swDocSpec = swApp.GetOpenDocSpec(compPath) as
IDocumentSpecification;
                if (swDocSpec == null)
                {
                    Console.WriteLine("[ERROR] Failed to get document specification for: " +
compPath);
                    skippedCount++;
                    continue;
                }

                swDocSpec.Silent = true;

                Console.WriteLine("[DEBUG] Opening document: " + compPath);
                IModelDoc2 swPartModel = swApp.OpenDoc7(swDocSpec) as IModelDoc2;

                if (swPartModel == null)
                {
                    Console.WriteLine("[ERROR] Failed to open part: " + compPath + " (Error: " +
swDocSpec.Error + ")");
                    skippedCount++;
                    continue;
                }

                Console.WriteLine("[DEBUG] Document opened successfully");

                // Activate the document
                int activateErrors = 0;
                swPartModel = swApp.ActivateDoc3(swPartModel.GetTitle(), false,
(int)swRebuildOnActivation_e.swUserDecision, ref activateErrors) as IModelDoc2;

                if (swPartModel == null)
                {
                    Console.WriteLine("[ERROR] Failed to activate part document. Error code: "
+ activateErrors);
                    skippedCount++;
                    continue;
                }
```

```
            Console.WriteLine("[DEBUG] Document activated successfully");

            // Cast to part document
            IPartDoc swPart = swPartModel as IPartDoc;
            if (swPart == null)
            {
                Console.WriteLine("[ERROR] Failed to cast to part document");
                swApp.CloseDoc(swPartModel.GetTitle());
                skippedCount++;
                continue;
            }

            // Check if this is a sheet metal part by looking for sheet metal features
            bool isSheetMetal = CheckIfSheetMetalPart(swPartModel);

            if (!isSheetMetal)
            {
                Console.WriteLine("[INFO] Part is not a sheet metal part, skipping: " +
System.IO.Path.GetFileName(compPath));
                swApp.CloseDoc(swPartModel.GetTitle());
                skippedCount++;
                continue;
            }

            Console.WriteLine("[SUCCESS] Confirmed sheet metal part: " +
System.IO.Path.GetFileName(compPath));

            // Prepare DXF export path
            string partName = System.IO.Path.GetFileNameWithoutExtension(compPath);
            string dxfPath = System.IO.Path.Combine(dxfOutputDir, partName +
"_FlatPattern.dxf");

            Console.WriteLine("[DEBUG] Exporting flat pattern to DXF: " + dxfPath);

            // Make the part visible for export
            swPartModel.Visible = true;

            // Export to DXF using ExportToDWG2 method with sheet metal options
            // This is the key method from VBA examples, converted to C#
            bool exportResult = swPart.ExportToDWG2(
                dxfPath,                    // Output file path
                compPath,                   // Model path
                (int)swExportToDWG_e.swExportToDWG_ExportSheetMetal,  // Export type
                true,                       // Export annotations
                null,                       // View names (empty for sheet metal)
                false,                      // Export views only
                false,                      // Export spline as spline
```

```
                    (int)SheetMetalOptions_e.ExportFlatPatternGeometry +
(int)SheetMetalOptions_e.ExportBendLines,  // Sheet metal options
                    null                    // Sheet metal bend lines
                );

                // Hide the part again
                swPartModel.Visible = false;

                if (exportResult)
                {
                    Console.WriteLine("[SUCCESS] Exported flat pattern: " + partName + " -> " +
dxfPath);
                    exportedCount++;
                    exportedFiles.Add(dxfPath);
                }
                else
                {
                    Console.WriteLine("[ERROR] Failed to export flat pattern for: " + partName);
                    skippedCount++;
                }

                // Close the part document
                swApp.CloseDoc(swPartModel.GetTitle());
                Console.WriteLine("[DEBUG] Closed part document: " + partName);
            }
            catch (Exception compEx)
            {
                Console.WriteLine("[ERROR] Exception processing component " +
swComp.Name2 + ": " + compEx.Message);
                skippedCount++;
            }
        }

        // Reactivate the original assembly
        int finalErrors = 0;
        swApp.ActivateDoc3(swModel.GetTitle(), false,
(int)swRebuildOnActivation_e.swUserDecision, ref finalErrors);

        Console.WriteLine("[COMPLETE] Bulk DXF export finished");
        Console.WriteLine("[RESULTS] Exported: " + exportedCount + ", Skipped: " +
skippedCount);
        Console.WriteLine("[INFO] DXF files saved to: " + dxfOutputDir);

        return new {
            success = true,
            message = "Successfully exported " + exportedCount + " sheet metal parts to DXF
flat pattern files. " + skippedCount + " components were skipped (non-sheet metal or
duplicates).",
```

```
            exported = exportedCount,
            skipped = skippedCount,
            outputDirectory = dxfOutputDir,
            exportedFiles = exportedFiles.ToArray()
        };
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Main exception: " + ex.Message);
        return new { success = false, error = "Main execution error: " + ex.Message };
    }
}

private static bool CheckIfSheetMetalPart(IModelDoc2 swModel)
{
    try
    {
        Console.WriteLine("[DEBUG] Checking if part is sheet metal");

        // Get feature manager
        IFeatureManager swFeatMgr = swModel.FeatureManager as IFeatureManager;
        if (swFeatMgr == null)
        {
            return false;
        }

        // Get first feature and traverse through all features
        IFeature swFeature = swModel.FirstFeature() as IFeature;

        while (swFeature != null)
        {
            string featType = swFeature.GetTypeName2();
            Console.WriteLine("[DEBUG] Checking feature: " + swFeature.Name + " (Type: " +
featType + ")");

            // Look for sheet metal specific features
            if (featType.Contains("SheetMetal") ||
                featType.Equals("BaseFlange") ||
                featType.Equals("EdgeFlange") ||
                featType.Equals("Hem") ||
                featType.Equals("Jog") ||
                featType.Equals("Fold") ||
                featType.Equals("Unfold") ||
                featType.Equals("FlatPattern") ||
                featType.Equals("ProcessBends"))
            {
                Console.WriteLine("[SUCCESS] Found sheet metal feature: " + featType);
                return true;
```

```
            }

            swFeature = swFeature.GetNextFeature() as IFeature;
        }

        Console.WriteLine("[INFO] No sheet metal features found");
        return false;
    }
    catch (Exception ex)
    {
        Console.WriteLine("[ERROR] Error checking sheet metal features: " + ex.Message);
        return false;
    }
  }
}
```

GuruAbhinai

Automation title : **Change part file names by changing custom property.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swconst;
using System;

    public static object RenamePartFromProperty()
    {
        try
        {
            Console.WriteLine("[START] Beginning part file rename based on custom property");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;
            if (swApp == null)
                return new { success = false, error = "Could not connect to SolidWorks application"
};

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get active document
            IModelDoc2 swModel = swApp.IActiveDoc2;
            if (swModel == null)
                return new { success = false, error = "No document is open. Please open a part
file." };

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
                return new { success = false, error = "Active document must be a part. Please
open a part document." };

            string currentFilePath = swModel.GetPathName();
            if (string.IsNullOrEmpty(currentFilePath))
                return new { success = false, error = "Document must be saved first. Please save
before running macro." };

            Console.WriteLine("[INFO] Current file path: " + currentFilePath);
```

```
        // Custom property manager
        ICustomPropertyManager swCustPropMgr =
swModel.Extension.get_CustomPropertyManager("") as ICustomPropertyManager;
        if (swCustPropMgr == null)
            return new { success = false, error = "Failed to get custom property manager" };

        string[] propertyNames = swCustPropMgr.GetNames() as string[];
        string selectedProperty = "";

        // If no custom props exist → create one
        if (propertyNames == null || propertyNames.Length == 0)
        {
            bool createProperty = ShowYesNoDialog("No custom properties found. Create
one?", "Create Custom Property");
            if (!createProperty) return new { success = false, error = "Cancelled - no custom
property", cancelled = true };

            string propertyName = ShowInputDialog("Enter custom property name:", "Custom
Property Name", "PartName");
            if (string.IsNullOrEmpty(propertyName)) return new { success = false, error = "No
property name", cancelled = true };

            string propertyValue = ShowInputDialog("Enter property value (will be new
filename):", "Property Value", "NewPartName");
            if (string.IsNullOrEmpty(propertyValue)) return new { success = false, error = "No
property value", cancelled = true };

            swCustPropMgr.Add3(propertyName,
(int)swCustomInfoType_e.swCustomInfoText, propertyValue,
(int)swCustomPropertyAddOption_e.swCustomPropertyReplaceValue);
            selectedProperty = propertyName;

            // Save doc to persist property
            int saveErr = 0, saveWarn = 0;
            swModel.Save3((int)swSaveAsOptions_e.swSaveAsOptions_Silent, ref saveErr,
ref saveWarn);
        }
        else
        {
            selectedProperty = ShowPropertySelectionDialog(propertyNames, "Select Custom
Property for File Name");
            if (string.IsNullOrEmpty(selectedProperty))
                return new { success = false, error = "No property selected", cancelled = true };
        }

        // Get property value
        string evaluatedValue, retrievedValue;
```

```
        bool wasResolved = swCustPropMgr.Get3(selectedProperty, false, out
evaluatedValue, out retrievedValue);
        if (!wasResolved || string.IsNullOrEmpty(evaluatedValue))
            return new { success = false, error = "Invalid or empty value for property: " +
selectedProperty };

        string cleanedValue = CleanFileNameString(evaluatedValue);
        if (string.IsNullOrEmpty(cleanedValue))
            return new { success = false, error = "Property value contains only invalid
characters" };

        string directory = System.IO.Path.GetDirectoryName(currentFilePath);
        string extension = System.IO.Path.GetExtension(currentFilePath);
        string newFilePath = System.IO.Path.Combine(directory, cleanedValue + extension);

        // Check if file already exists
        if (System.IO.File.Exists(newFilePath) && !string.Equals(currentFilePath,
newFilePath, StringComparison.OrdinalIgnoreCase))
        {
            bool overwrite = ShowYesNoDialog("File already exists: " + cleanedValue +
extension + ". Overwrite?", "File Exists");
            if (!overwrite) return new { success = false, error = "Rename cancelled - file
exists", cancelled = true };
        }

        // Save with new name
        int finalErr = 0, finalWarn = 0;
        bool saveResult = swModel.Extension.SaveAs(newFilePath,
            (int)swSaveAsVersion_e.swSaveAsCurrentVersion,
            (int)swSaveAsOptions_e.swSaveAsOptions_Silent,
            null, ref finalErr, ref finalWarn);

        if (!saveResult)
            return new { success = false, error = "Failed to save as new file. Error: " + finalErr
};

        // Ask if close old doc
        bool closeOriginal = ShowYesNoDialog("File renamed successfully! Close original
file?", "Close Original");
        if (closeOriginal) swApp.CloseDoc(currentFilePath);

        Console.WriteLine("[SUCCESS] File renamed successfully");
        return new {
            success = true,
            message = "File renamed to \"" + System.IO.Path.GetFileName(newFilePath) + "\"
based on property \"" + selectedProperty + "\"",
            originalFile = System.IO.Path.GetFileName(currentFilePath),
            newFile = System.IO.Path.GetFileName(newFilePath),
```

```
            propertyUsed = selectedProperty,
            propertyValue = evaluatedValue,
            newFilePath = newFilePath
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Exception: " + ex.Message);
        return new { success = false, error = "Exception: " + ex.Message };
    }
}

// Utility dialogs + helpers (ShowInputDialog, ShowPropertySelectionDialog,
ShowYesNoDialog, CleanFileNameString)
// ... [identiques à ton code initial, inchangés]
```

RottenVagy

Automation title : **Create a drawing view for every unique solid body in a part file, add view label with linked bom item number + qty + view scale. Sister macro to re-arrange these views to be in same order as bom.**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

```
public class CreateDrawing
{
    public static object Drawing()
    {
        try
        {
            Console.WriteLine(
                "[START] Creating ONE drawing with MULTIPLE SHEETS - each sheet showing
ONE body"
            );

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
                Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application"))
                as SldWorks;

            if (swApp == null)
            {
                return new
                {
                    success = false,
                    error = "Could not connect to SolidWorks application",
                };
            }

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;

            if (swModel == null)
            {
```

```
            return new
            {
                success = false,
                error = "No document is open. Please open a part document with solid bodies.",
            };
        }

        Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

        // Check if the active document is a part
        if (swModel.GetType() != (int)swDocumentTypes_e.swDocPART)
        {
            return new
            {
                success = false,
                error = "Active document must be a part. Please open a part document with
multiple solid bodies.",
            };
        }

        IPartDoc swPart = swModel as IPartDoc;
        if (swPart == null)
        {
            return new { success = false, error = "Could not cast to IPartDoc interface" };
        }

        Console.WriteLine("[INFO] Part document confirmed");

        // Get all solid bodies
        Console.WriteLine("[DEBUG] Getting solid bodies from part...");
        object[] bodies =
            swPart.GetBodies2((int)swBodyType_e.swSolidBody, true) as object[];

        if (bodies == null || bodies.Length == 0)
        {
            Console.WriteLine("[ERROR] GetBodies2 returned null or empty array");
            return new { success = false, error = "No solid bodies found in the part" };
        }

        Console.WriteLine("[INFO] Found " + bodies.Length + " solid bodies in the part");

        // Debug: Show raw bodies from GetBodies2
        Console.WriteLine("[DEBUG] Raw bodies from GetBodies2:");
        for (int rawIndex = 0; rawIndex < bodies.Length; rawIndex++)
        {
            IBody2 rawBody = bodies[rawIndex] as IBody2;
            string rawName = rawBody != null ? rawBody.Name : "NULL";
            Console.WriteLine(
```

```
                "[DEBUG] Raw body "
                    + (rawIndex + 1)
                    + ": '"
                    + rawName
                    + "' (Valid: "
                    + (rawBody != null)
                    + ")"
                );
            }

            // Process all solid bodies (each body is treated as unique regardless of name)
            System.Collections.Generic.List<IBody2> allBodies =
                new System.Collections.Generic.List<IBody2>();
            System.Collections.Generic.List<string> bodyDisplayNames =
                new System.Collections.Generic.List<string>();
            // Persist references survive rebuilds/config switches; re-resolve per config to avoid
stale IBody2
            System.Collections.Generic.List<object> bodyPersistReferences =
                new System.Collections.Generic.List<object>();

            Console.WriteLine(
                "[DEBUG] Processing " + bodies.Length + " bodies into allBodies list..."
            );

            for (int i = 0; i < bodies.Length; i++)
            {
                Console.WriteLine(
                    "[DEBUG] Processing raw body " + (i + 1) + " of " + bodies.Length
                );

                IBody2 swBody = bodies[i] as IBody2;
                if (swBody == null)
                {
                    Console.WriteLine("[WARNING] Body " + (i + 1) + " is NULL, skipping");
                    continue;
                }

                // Create a unique display name for each body
                string originalName = swBody.Name;
                string displayName;

                if (string.IsNullOrEmpty(originalName))
                {
                    displayName = "Body" + (i + 1);
                }
                else
                {
                    // If multiple bodies have the same name, append index
```

```
            displayName = originalName;
            int nameCount = 1;
            string testName = displayName;
            while (bodyDisplayNames.Contains(testName))
            {
                nameCount++;
                testName = displayName + "_" + nameCount;
            }
            displayName = testName;
        }

        allBodies.Add(swBody);
        bodyDisplayNames.Add(displayName);
        // Capture a persist reference for reliable re-resolution later
        object persistRef = swModel.Extension.GetPersistReference3(swBody);
        bodyPersistReferences.Add(persistRef);
        Console.WriteLine(
            "[INFO] Added body "
                + (i + 1)
                + ": "
                + displayName
                + " (original name: '"
                + (originalName ?? "empty")
                + "')"
        );
        Console.WriteLine("[DEBUG] allBodies.Count is now: " + allBodies.Count);
    }

    Console.WriteLine("[DEBUG] Finished processing all raw bodies");
    Console.WriteLine(
        "[DEBUG] Final counts - Raw bodies: "
            + bodies.Length
            + ", Processed bodies: "
            + allBodies.Count
    );

    if (allBodies.Count == 0)
    {
        return new { success = false, error = "No solid bodies found" };
    }

    Console.WriteLine("[INFO] Found " + allBodies.Count + " solid bodies to process");

    // Debug: Show all bodies that will be processed
    Console.WriteLine("[DEBUG] Bodies to process:");
    for (int debugIndex = 0; debugIndex < allBodies.Count; debugIndex++)
    {
        IBody2 debugBody = allBodies[debugIndex];
```

```
    string debugName = bodyDisplayNames[debugIndex];
    Console.WriteLine(
        "[DEBUG] Body "
            + (debugIndex + 1)
            + ": "
            + debugName
            + " (Body object: "
            + (debugBody != null ? "Valid" : "NULL")
            + ")"
    );
}

// STEP 1: Create configurations for each body (using KeepOnly approach)
Console.WriteLine("[STEP 1] Creating configurations for each body");

IConfigurationManager swConfigMgr = swModel.ConfigurationManager;
IConfiguration swActiveConfig = swConfigMgr.ActiveConfiguration;
string baseConfigName = swActiveConfig.Name;

Console.WriteLine("[INFO] Base configuration: " + baseConfigName);

System.Collections.Generic.List<string> createdConfigs =
    new System.Collections.Generic.List<string>();
int keepOnlyFeaturesCreated = 0;

// Create configuration for each body
Console.WriteLine(
    "[DEBUG] Starting configuration creation loop for "
        + allBodies.Count
        + " bodies"
);

for (int bodyIndex = 0; bodyIndex < allBodies.Count; bodyIndex++)
{
    Console.WriteLine(
        "[DEBUG] ========== Processing body "
            + (bodyIndex + 1)
            + " of "
            + allBodies.Count
            + " =========="
    );

    IBody2 swBody = allBodies[bodyIndex];
    string bodyName = bodyDisplayNames[bodyIndex];

    Console.WriteLine(
        "[DEBUG] Body index: "
            + bodyIndex
```

```
        + ", Body name: "
        + bodyName
        + ", Body valid: "
        + (swBody != null)
);

string configName =
    "Config_"
    + bodyName
        .Replace(" ", "_")
        .Replace("-", "_")
        .Replace(".", "_")
        .Replace("[", "")
        .Replace("]", "");

Console.WriteLine(
    "[DEBUG] Creating configuration for body: " + bodyName + " -> " + configName
);

try
{
    // Check if configuration already exists
    string[] existingConfigs = swModel.GetConfigurationNames() as string[];
    bool configExists = false;
    if (existingConfigs != null)
    {
        foreach (string existingConfig in existingConfigs)
        {
            if (existingConfig == configName)
            {
                configExists = true;
                Console.WriteLine(
                    "[INFO] Configuration " + configName + " already exists"
                );
                break;
            }
        }
    }

    IConfiguration swBodyConfig = null;

    if (!configExists)
    {
        // Create new configuration based on the active configuration
        swBodyConfig = swConfigMgr.AddConfiguration2(
            configName,
            "Configuration showing only " + bodyName,
            "",
```

```
            (int)swConfigurationOptions2_e.swConfigOption_DontActivate,
            baseConfigName,
            "",
            false
        );

        if (swBodyConfig == null)
        {
            Console.WriteLine(
                "[ERROR] Failed to create configuration for body: " + bodyName
            );
            continue;
        }

        Console.WriteLine("[SUCCESS] Created configuration: " + configName);
    }

    // Activate the configuration to work in it
    Console.WriteLine("[DEBUG] Activating configuration: " + configName);
    bool activated = swModel.ShowConfiguration2(configName);

    if (!activated)
    {
        Console.WriteLine(
            "[ERROR] Failed to activate configuration: " + configName
        );
        continue;
    }

    // Clear any existing selections
    swModel.ClearSelection2(true);

    // Create array of all OTHER bodies (bodies to delete, keeping only current one)
    System.Collections.Generic.List<IBody2> bodiesToDelete =
        new System.Collections.Generic.List<IBody2>();

    // Re-resolve live IBody2 objects in the CURRENT configuration from persist
refs
    for (int j = 0; j < allBodies.Count; j++)
    {
        if (j == bodyIndex)
        {
            continue; // keep this one
        }

        int objType = 0;
        object resolved = swModel.Extension.GetObjectByPersistReference3(
            bodyPersistReferences[j],
```

```
            out objType
        );

        IBody2 bodyToDelete = resolved as IBody2;
        if (bodyToDelete != null)
        {
            bodiesToDelete.Add(bodyToDelete);
        }
        else
        {
            Console.WriteLine(
                "[WARNING] Failed to resolve body from persist ref at index "
                    + j
            );
        }
    }

    Console.WriteLine(
        "[DEBUG] Will delete "
            + bodiesToDelete.Count
            + " bodies, keeping: "
            + bodyName
    );

    if (bodiesToDelete.Count > 0)
    {
        Console.WriteLine(
            "[DEBUG] About to create KeepOnly feature for body: " + bodyName
        );
        Console.WriteLine(
            "[DEBUG] Will delete " + bodiesToDelete.Count + " other bodies"
        );

        // Convert list to array for selection
        IBody2[] deleteArray = bodiesToDelete.ToArray();

        Console.WriteLine(
            "[DEBUG] Selecting " + deleteArray.Length + " bodies for deletion"
        );

        // Select all bodies to delete
        int selectedCount = swModel.Extension.MultiSelect2(
            deleteArray,
            false,
            null
        );

        if (selectedCount == deleteArray.Length)
```

```
{
    Console.WriteLine(
        "[DEBUG] Successfully selected " + selectedCount + " bodies"
    );

    // Create Delete Body feature (KeepOnly for remaining body)
    IFeature swDeleteBodyFeature =
        swModel.FeatureManager.InsertDeleteBody2(false);

    if (swDeleteBodyFeature != null)
    {
        string keepOnlyFeatureName =
            "KeepOnly_" + bodyName.Replace("[", "").Replace("]", "");
        swDeleteBodyFeature.Name = keepOnlyFeatureName;

        keepOnlyFeaturesCreated++;
        Console.WriteLine(
            "[SUCCESS] Created KeepOnly feature: "
                + swDeleteBodyFeature.Name
                + " for body: "
                + bodyName
                + " (bodyIndex: "
                + bodyIndex
                + ")"
                + " [Total KeepOnly features created: "
                + keepOnlyFeaturesCreated
                + "]"
        );

        // Suppress this feature in the base configuration
        string[] baseConfigArray = new string[] { baseConfigName };
        bool suppressResult = swDeleteBodyFeature.SetSuppression2(
            (int)swFeatureSuppressionAction_e.swSuppressFeature,
            (int)swInConfigurationOpts_e.swSpecifyConfiguration,
            baseConfigArray
        );

        // Unsuppress the feature in the current body configuration
        string[] bodyConfigArray = new string[] { configName };
        bool unsuppressResult = swDeleteBodyFeature.SetSuppression2(
            (int)swFeatureSuppressionAction_e.swUnSuppressFeature,
            (int)swInConfigurationOpts_e.swSpecifyConfiguration,
            bodyConfigArray
        );

        Console.WriteLine(
            "[DEBUG] Set suppression states for KeepOnly feature"
        );
```

```
                    }
                    else
                    {
                        Console.WriteLine(
                            "[ERROR] Failed to create KeepOnly feature for " + bodyName
                        );
                    }
                }
                else
                {
                    Console.WriteLine(
                        "[ERROR] Could only select "
                            + selectedCount
                            + " out of "
                            + deleteArray.Length
                            + " bodies"
                    );
                }

                // Clear selections
                swModel.ClearSelection2(true);
            }

            createdConfigs.Add(configName);
            Console.WriteLine(
                "[SUCCESS] Completed setup for configuration: " + configName
            );
            Console.WriteLine(
                "[DEBUG] ========== Finished processing body "
                    + (bodyIndex + 1)
                    + " =========="
            );
        }
        catch (System.Exception configEx)
        {
            Console.WriteLine(
                "[ERROR] Exception creating configuration for "
                    + bodyName
                    + ": "
                    + configEx.Message
            );
            Console.WriteLine(
                "[DEBUG] ========== ERROR in body " + (bodyIndex + 1) + "
==========="
            );
        }
    }
```

```
        Console.WriteLine(
            "[DEBUG] Configuration creation loop completed. Created "
                + createdConfigs.Count
                + " configurations"
        );
        Console.WriteLine(
            "[DEBUG] TOTAL KeepOnly features created: "
                + keepOnlyFeaturesCreated
                + " (Expected: "
                + allBodies.Count
                + ")"
        );

        // Return to base configuration
        Console.WriteLine("[INFO] Returning to base configuration: " + baseConfigName);
        swModel.ShowConfiguration2(baseConfigName);
        swModel.ForceRebuild3(false);

        Console.WriteLine(
            "[STEP 1 COMPLETE] Created " + createdConfigs.Count + " configurations"
        );

        // STEP 2: Create drawing document with multiple sheets
        Console.WriteLine("[STEP 2] Creating drawing document with multiple sheets");

        string drawingTemplate =
            "C:\\ProgramData\\SOLIDWORKS\\SOLIDWORKS 2025\\templates\\Mise en
plan.DRWDOT";
        Console.WriteLine("[INFO] Using drawing template: " + drawingTemplate);

        // Verify template exists
        if (!System.IO.File.Exists(drawingTemplate))
        {
            Console.WriteLine(
                "[ERROR] Template file does not exist at: " + drawingTemplate
            );
            return new
            {
                success = false,
                error = "Drawing template file not found at specified path: "
                    + drawingTemplate,
            };
        }

        // Create drawing document name for saving
        string partPath = swModel.GetPathName();
        string drawingPath = "";
```

```
        if (string.IsNullOrEmpty(partPath))
        {
            drawingPath = System.IO.Path.Combine(
                System.IO.Path.GetTempPath(),
                "TempDrawing_" + System.DateTime.Now.ToString("yyyyMMdd_HHmmss") +
".slddrw"
            );
        }
        else
        {
            string directory = System.IO.Path.GetDirectoryName(partPath);
            string baseFileName = System.IO.Path.GetFileNameWithoutExtension(partPath);
            drawingPath = System.IO.Path.Combine(
                directory,
                baseFileName + "_Bodies.slddrw"
            );
        }

        Console.WriteLine("[INFO] Will create drawing at: " + drawingPath);

        // Create new drawing document
        IModelDoc2 swDrawingModel = null;

        try
        {
            swDrawingModel =
                swApp.INewDocument2(
                    drawingTemplate,
                    (int)swDwgPaperSizes_e.swDwgPaperA3size,
                    0.0,
                    0.0
                ) as IModelDoc2;
            Console.WriteLine("[INFO] Created drawing document successfully");
        }
        catch (System.Exception createEx)
        {
            Console.WriteLine("[ERROR] Exception creating drawing: " + createEx.Message);
            return new
            {
                success = false,
                error = "Failed to create drawing: " + createEx.Message,
            };
        }

        if (swDrawingModel == null)
        {
            return new { success = false, error = "Failed to create new drawing document" };
        }
```

```
        IDrawingDoc swDrawing = swDrawingModel as IDrawingDoc;
        if (swDrawing == null)
        {
            return new
            {
                success = false,
                error = "Could not cast to IDrawingDoc interface",
            };
        }

        // Save drawing document
        try
        {
            bool saveResult = swDrawingModel.SaveAs(drawingPath);
            if (saveResult)
            {
                Console.WriteLine("[INFO] Drawing saved to: " + drawingPath);
            }
        }
        catch (System.Exception saveEx)
        {
            Console.WriteLine("[WARNING] Error saving drawing: " + saveEx.Message);
        }

        // Get the first (default) sheet
        ISheet swFirstSheet = swDrawing.GetCurrentSheet() as ISheet;
        if (swFirstSheet == null)
        {
            return new { success = false, error = "Could not get current drawing sheet" };
        }

        Console.WriteLine("[INFO] Working with first sheet: " + swFirstSheet.GetName());

        // STEP 3: Create views and sheets for each body configuration
        Console.WriteLine("[STEP 3] Creating views and sheets for each body
configuration");

        int sheetsCreated = 0;
        System.Collections.Generic.List<string> createdSheetNames =
            new System.Collections.Generic.List<string>();

        for (int bodyIndex = 0; bodyIndex < allBodies.Count; bodyIndex++)
        {
            string bodyName = bodyDisplayNames[bodyIndex];
            string configName = createdConfigs[bodyIndex];

            Console.WriteLine(
```

```
    "[INFO] Creating sheet and view for body: "
        + bodyName
        + " using config: "
        + configName
);

try
{
    ISheet targetSheet = null;

    if (bodyIndex == 0)
    {
        // Use the first sheet for the first body
        targetSheet = swFirstSheet;
        targetSheet.SetName("Sheet_" + bodyName);
        Console.WriteLine(
            "[INFO] Using first sheet, renamed to: Sheet_" + bodyName
        );
    }
    else
    {
        // Create new sheet for subsequent bodies by copying the first sheet
        Console.WriteLine("[DEBUG] Copying sheet for body: " + bodyName);

        // Select the first sheet for copying
        bool sheetSelected = swDrawingModel.Extension.SelectByID2(
            swFirstSheet.GetName(),
            "SHEET",
            0,
            0,
            0,
            false,
            0,
            null,
            0
        );
        if (sheetSelected)
        {
            swDrawingModel.EditCopy();

            // Paste the sheet
            bool pasteResult = swDrawing.PasteSheet(
                (int)swInsertOptions_e.swInsertOption_MoveToEnd,
                (int)swRenameOptions_e.swRenameOption_Yes
            );

            if (pasteResult)
            {
```

```
            // Get the newly created sheet (last one)
            string[] sheetNames = swDrawing.GetSheetNames() as string[];
            string newSheetName = sheetNames[sheetNames.Length - 1];
            targetSheet = swDrawing.get_Sheet(newSheetName) as ISheet;

            if (targetSheet != null)
            {
                targetSheet.SetName("Sheet_" + bodyName);
                Console.WriteLine(
                    "[SUCCESS] Created new sheet: Sheet_" + bodyName
                );
            }
            else
            {
                Console.WriteLine(
                    "[ERROR] Could not get newly created sheet"
                );
                continue;
            }
        }
        else
        {
            Console.WriteLine(
                "[ERROR] Failed to paste sheet for body: " + bodyName
            );
            continue;
        }
    }
    else
    {
        Console.WriteLine("[ERROR] Failed to select sheet for copying");
        continue;
    }
}

if (targetSheet == null)
{
    Console.WriteLine(
        "[ERROR] No target sheet available for body: " + bodyName
    );
    continue;
}

// Activate the target sheet
swDrawing.ActivateSheet(targetSheet.GetName());
Console.WriteLine("[DEBUG] Activated sheet: " + targetSheet.GetName());

// If this is a copied sheet (not the first one), delete any existing views
```

```
if (bodyIndex > 0)
{
    Console.WriteLine("[DEBUG] Cleaning up views on copied sheet");

    // Get all views on this sheet
    object[] existingViews = targetSheet.GetViews() as object[];
    if (existingViews != null && existingViews.Length > 0)
    {
        Console.WriteLine(
            "[DEBUG] Found "
                + existingViews.Length
                + " existing views on copied sheet"
        );

        // Delete all existing drawing views
        for (
            int viewIdx = existingViews.Length - 1;
            viewIdx >= 0;
            viewIdx--
        )
        {
            IView existingView = existingViews[viewIdx] as IView;
            if (existingView != null)
            {
                Console.WriteLine(
                    "[DEBUG] Deleting copied view: "
                        + existingView.GetName2()
                );

                // Select and delete the view
                bool viewSelected = swDrawingModel.Extension.SelectByID2(
                    existingView.GetName2(),
                    "DRAWINGVIEW",
                    0,
                    0,
                    0,
                    false,
                    0,
                    null,
                    0
                );

                if (viewSelected)
                {
                    swDrawingModel.Extension.DeleteSelection2(
                        (int)swDeleteSelectionOptions_e.swDelete_Absorbed
                    );
                    Console.WriteLine(
```

```
                    "[DEBUG] Successfully deleted copied view"
                );
            }
            else
            {
                Console.WriteLine(
                    "[WARNING] Could not select view for deletion: "
                        + existingView.GetName2()
                );
            }
        }
    }

    // Clear selections after cleanup
    swDrawingModel.ClearSelection2(true);
    Console.WriteLine("[DEBUG] Finished cleaning up copied sheet");
  }
}

// Clear any existing selections
swDrawingModel.ClearSelection2(true);

// Create view position
double viewX = 0.15; // 150mm from left
double viewY = 0.15; // 150mm from bottom

Console.WriteLine(
  "[DEBUG] Creating view at position: X=" + viewX + ", Y=" + viewY
);

// Create drawing view referencing the specific configuration
IView swView = swDrawing.CreateDrawViewFromModelView3(
  partPath, // Model path
  "*Isometric", // View name (isometric)
  viewX, // X position
  viewY, // Y position
  0 // Z position
);

if (swView != null)
{
  Console.WriteLine(
    "[DEBUG] Created view: "
      + swView.GetName2()
      + ", setting configuration: "
      + configName
  );
```

```
                // CRITICAL: Set the configuration to show only this body
                swView.ReferencedConfiguration = configName;

                // Force immediate update by refreshing the drawing
                swDrawingModel.GraphicsRedraw2();

                // Force view to update with the new configuration
                Console.WriteLine("[DEBUG] Refreshing view to apply configuration");

                // Force the view to update with the new configuration
                // Use a more reliable approach: rebuild the drawing model
                swDrawingModel.EditRebuild3();

                // Additional step: force view to update by setting scale again
                // This triggers an internal update of the view
                double[] currentScale = swView.ScaleRatio as double[];
                if (currentScale != null)
                {
                    swView.ScaleRatio = currentScale; // Re-setting triggers refresh
                }

                Console.WriteLine("[DEBUG] View configuration applied and refreshed");

                // Set view scale
                swView.ScaleRatio = new double[] { 1, 2 }; // 1:2 scale

                // Force another rebuild to ensure everything is updated
                // Note: IView doesn't have GraphicsRedraw method, using model rebuild
instead
                Console.WriteLine(
                    "[SUCCESS] Created view for body: "
                        + bodyName
                        + " with config: "
                        + configName
                );

                // Create label for this view
                string labelText =
                    "Item: "
                    + (bodyIndex + 1)
                    + "\nQty: 1\nScale: 1:2\nBody: "
                    + bodyName;

                // Position label below the view
                double labelX = viewX;
                double labelY = viewY - 0.05; // 50mm below view center

                // Create note annotation
```

```
            Console.WriteLine(
                "[DEBUG] Creating label at position: X=" + labelX + ", Y=" + labelY
            );

            // Clear selections before inserting note
            swDrawingModel.ClearSelection2(true);

            // Create note
            INote swNote = swDrawingModel.InsertNote(labelText) as INote;
            if (swNote != null)
            {
                IAnnotation swAnnotation = swNote.GetAnnotation() as IAnnotation;
                if (swAnnotation != null)
                {
                    // Set note position
                    swAnnotation.SetPosition2(labelX, labelY, 0);

                    // Set text formatting
                    ITextFormat swTextFormat =
                        swNote.GetTextFormat() as ITextFormat;
                    if (swTextFormat != null)
                    {
                        swTextFormat.CharHeight = 0.003; // 3mm text height
                        swTextFormat.Bold = true;
                    }

                    Console.WriteLine("[INFO] Created label for body: " + bodyName);
                }
            }

            sheetsCreated++;
            createdSheetNames.Add("Sheet_" + bodyName);
        }
        else
        {
            Console.WriteLine(
                "[ERROR] Failed to create drawing view for body: " + bodyName
            );
        }
    }
    catch (System.Exception viewEx)
    {
        Console.WriteLine(
            "[ERROR] Exception creating view for body "
                + bodyName
                + ": "
                + viewEx.Message
        );
```

```
    }
}

// Final rebuild and refresh of all views
Console.WriteLine("[INFO] Final rebuild and refresh of all drawing views");

// Go through each created sheet and ensure it's properly updated
string[] allSheetNames = swDrawing.GetSheetNames() as string[];
if (allSheetNames != null)
{
    foreach (string sheetName in allSheetNames)
    {
        if (createdSheetNames.Contains(sheetName))
        {
            Console.WriteLine("[DEBUG] Final refresh of sheet: " + sheetName);

            // Activate the sheet
            swDrawing.ActivateSheet(sheetName);

            // Force sheet rebuild to ensure all views show correct configurations
            swDrawingModel.EditRebuild3();
            swDrawingModel.GraphicsRedraw2();

            // Small delay to allow SolidWorks to process the updates
            System.Threading.Thread.Sleep(100);
        }
    }
}

// Final global rebuild
Console.WriteLine("[INFO] Final global rebuild of drawing");
swDrawingModel.ForceRebuild3(false);
swDrawingModel.GraphicsRedraw2();

// Zoom to fit
swDrawingModel.ViewZoomtofit2();

Console.WriteLine("[COMPLETE] Drawing creation finished");
Console.WriteLine(
    "[RESULTS] Successfully created "
        + sheetsCreated
        + " sheets with individual body views"
);

return new
{
    success = true,
    message = "Successfully created "
```

```
                + sheetsCreated
                + " drawing sheets, each showing one unique solid body! Each sheet contains an isometric view of only one body with proper labeling (item number, quantity, scale). The drawing is saved at: "
                + drawingPath,
            totalBodies = allBodies.Count,
            sheetsCreated = sheetsCreated,
            configurationsCreated = createdConfigs.Count,
            drawingPath = drawingPath,
            sheetNames = createdSheetNames,
            bodyNames = bodyDisplayNames,
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Main exception: " + ex.Message);
        Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
        return new { success = false, error = "Exception: " + ex.Message };
    }
  }
}
```

Capibar2004

Automation title : **Drawing macro to extend hole centermarks arms. Good luck XD**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swconst;
using System;


    public static object AddAndExtendCenterMarks()
    {
        try
        {
            Console.WriteLine("[START] Beginning automatic centermark creation for all holes");

            // Create SolidWorks instance (will be fixed by executor to use
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;
            if (swApp == null)
                return new { success = false, error = "Could not connect to SolidWorks application"
};

            Console.WriteLine("[INFO] Connected to SolidWorks");

            // Get the active document
            IModelDoc2 swModel = swApp.IActiveDoc2;
            if (swModel == null)
                return new { success = false, error = "No document is open. Please open a
drawing document." };

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            if (swModel.GetType() != (int)swDocumentTypes_e.swDocDRAWING)
                return new { success = false, error = "Active document must be a drawing. Please
open a drawing document with holes." };

            IDrawingDoc swDrawing = swModel as IDrawingDoc;
            if (swDrawing == null)
                return new { success = false, error = "Failed to cast to drawing document" };

            Console.WriteLine("[INFO] Drawing document confirmed");
```

```
        ISheet currentSheet = swDrawing.GetCurrentSheet() as ISheet;
        if (currentSheet == null)
            return new { success = false, error = "No active sheet found in drawing" };

        Console.WriteLine("[INFO] Current sheet: " + currentSheet.GetName());

        object[] views = currentSheet.GetViews() as object[];
        if (views == null || views.Length == 0)
            return new { success = false, error = "No drawing views found on current sheet" };

        Console.WriteLine("[INFO] Found " + views.Length + " drawing views");

        int totalHolesFound = 0;
        int totalCentermarksAdded = 0;
        int totalArmsExtended = 0;

        // --- Phase 1: insert centermarks ---
        for (int v = 0; v < views.Length; v++)
        {
            IView view = views[v] as IView;
            if (view == null) continue;

            string viewName = view.GetName2();
            Console.WriteLine("[INFO] Processing view: " + viewName);

            if (viewName.ToLower().Contains("sheet"))
            {
                Console.WriteLine("[SKIP] Skipping sheet format view: " + viewName);
                continue;
            }

            try
            {
                swDrawing.ActivateView(viewName);
                bool autoInsertResult = view.AutoInsertCenterMarks2(
                    1, 0, false, false, true, 0.005, 0.001, true, false, 0.0
                );

                if (autoInsertResult)
                {
                    ICenterMark cm = view.GetFirstCenterMark() as ICenterMark;
                    int count = 0;
                    while (cm != null) { count++; cm = cm.GetNext() as ICenterMark; }
                    totalCentermarksAdded += count;
                    Console.WriteLine("[INFO] Added " + count + " centermarks automatically in
view: " + viewName);
                }
```

```
            }
            catch { }
        }

        Console.WriteLine("[PHASE 1 COMPLETE] Found " + totalHolesFound + " holes,
added " + totalCentermarksAdded + " centermarks");

        // --- Phase 2: extend centermark arms ---
        double extensionLength = 0.01; // 10 mm
        Console.WriteLine("[PHASE 2] Extending centermark arms...");

        for (int v = 0; v < views.Length; v++)
        {
            IView view = views[v] as IView;
            if (view == null) continue;

            string viewName = view.GetName2();
            if (viewName.ToLower().Contains("sheet")) continue;

            ICenterMark cm = view.GetFirstCenterMark() as ICenterMark;
            while (cm != null)
            {
                for (int handleId = 0; handleId <= 3; handleId++)
                {
                    double currentLength = cm.GetExtendedLength(0, handleId);
                    double newLength = currentLength + extensionLength;
                    if (cm.SetExtendedLength(0, handleId, newLength))
                        totalArmsExtended++;
                }
                cm = cm.GetNext() as ICenterMark;
            }
        }

        swModel.ForceRebuild3(false);
        swModel.GraphicsRedraw2();

        string summary = "Added " + totalCentermarksAdded + " centermark(s) and
extended "
                    + totalArmsExtended + " arms (" + (extensionLength * 1000).ToString("F1")
+ " mm each).";

        Console.WriteLine("[COMPLETE] " + summary);

        return new {
            success = true,
            message = summary,
            holesFound = totalHolesFound,
            centermarksAdded = totalCentermarksAdded,
```

```
            armsExtended = totalArmsExtended,
            extensionLength = (extensionLength * 1000).ToString("F1") + "mm"
        };
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Main exception: " + ex.Message);
        return new { success = false, error = "Exception during centermark creation: " +
ex.Message };
    }
}
```

Free_Koala_1629

Automation title : **macro demo Shortcut for save as… hotkey for stl save or step save etc sans son**

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

**Here is the code :**

```
using SolidWorks.Interop.sldworks;
using SolidWorks.Interop.swconst;
using System;

public class SaveAsShortcutMacro
{
    public static object RunSaveAsShortcut()
    {
        try
        {
            Console.WriteLine("[START] Beginning Save As Shortcut macro");

            // Create SolidWorks instance (will be fixed by executor with
Marshal.GetActiveObject)
            SldWorks swApp =
Activator.CreateInstance(Type.GetTypeFromProgID("SldWorks.Application")) as SldWorks;
            if (swApp == null)
                return new { success = false, error = "Could not connect to SolidWorks application"
};

            Console.WriteLine("[INFO] Connected to SolidWorks successfully");

            // Get active document
            IModelDoc2 swModel = swApp.IActiveDoc2;
            if (swModel == null)
                return new { success = false, error = "No document is open. Please open a part,
assembly, or drawing document." };

            Console.WriteLine("[INFO] Retrieved active document: " + swModel.GetTitle());

            // Check document type
            int docType = swModel.GetType();
            Console.WriteLine("[INFO] Document type: " +
                (docType == (int)swDocumentTypes_e.swDocPART ? "Part" :
                 docType == (int)swDocumentTypes_e.swDocASSEMBLY ? "Assembly" :
"Drawing"));
```

```
        // Ensure document is saved
        string currentPath = swModel.GetPathName();
        if (string.IsNullOrEmpty(currentPath))
            return new { success = false, error = "Please save the document first before using this export shortcut." };

        Console.WriteLine("[INFO] Document path: " + currentPath);

        string directory = System.IO.Path.GetDirectoryName(currentPath);
        string fileNameWithoutExt = System.IO.Path.GetFileNameWithoutExtension(currentPath);

        Console.WriteLine("[INFO] Export directory: " + directory);
        Console.WriteLine("[INFO] Base filename: " + fileNameWithoutExt);

        // Show format selection dialog
        Console.WriteLine("[INFO] Showing Save As format selection dialog...");
        string selectedFormat = "";
        string selectedExtension = "";
        bool userCancelled = false;

        using (var formatDialog = new SaveAsShortcutForm(docType))
        {
            var dialogResult = formatDialog.ShowDialog();
            if (dialogResult == System.Windows.Forms.DialogResult.OK)
            {
                selectedFormat = formatDialog.GetSelectedFormat();
                selectedExtension = formatDialog.GetSelectedExtension();
                Console.WriteLine("[INFO] User selected format: " + selectedFormat);
            }
            else
            {
                Console.WriteLine("[INFO] User cancelled format selection");
                userCancelled = true;
            }
        }

        if (userCancelled)
            return new { success = true, message = "Save As operation cancelled by user", result = "User cancelled" };

        // Create export file path
        string exportFilePath = System.IO.Path.Combine(directory, fileNameWithoutExt + selectedExtension);
        Console.WriteLine("[INFO] Export file path: " + exportFilePath);

        // Overwrite protection
        if (System.IO.File.Exists(exportFilePath))
```

```
        {
            var result = System.Windows.Forms.MessageBox.Show(
                "File already exists: " + System.IO.Path.GetFileName(exportFilePath) +
                "\n\nDo you want to overwrite it?",
                "File Exists",
                System.Windows.Forms.MessageBoxButtons.YesNo,
                System.Windows.Forms.MessageBoxIcon.Question
            );

            if (result != System.Windows.Forms.DialogResult.Yes)
                return new { success = true, message = "Export cancelled - file already exists",
result = "User cancelled overwrite" };
        }

        swModel.ClearSelection2(true);
        Console.WriteLine("[DEBUG] Cleared all selections for complete model export");

        // Perform the export
        Console.WriteLine("[DEBUG] Starting " + selectedFormat + " export...");
        bool exportSuccess = false;
        string errorMessage = "";

        if (selectedFormat == "STL")
            exportSuccess = ExportSTL(swModel, exportFilePath, out errorMessage);
        else if (selectedFormat == "STEP")
            exportSuccess = ExportSTEP(swModel, exportFilePath, out errorMessage);
        else if (selectedFormat == "PDF")
            exportSuccess = ExportPDF(swApp, swModel, exportFilePath, out errorMessage);
        else if (selectedFormat == "IGES")
            exportSuccess = ExportIGES(swModel, exportFilePath, out errorMessage);
        else if (selectedFormat == "Parasolid")
            exportSuccess = ExportParasolid(swModel, exportFilePath, out errorMessage);
        else
            return new { success = false, error = "Unsupported export format: " +
selectedFormat };

        if (exportSuccess)
        {
            if (System.IO.File.Exists(exportFilePath))
            {
                var fileInfo = new System.IO.FileInfo(exportFilePath);
                Console.WriteLine("[SUCCESS] " + selectedFormat + " export completed
successfully");
                Console.WriteLine("[INFO] File size: " + (fileInfo.Length / 1024) + " KB");

                return new
                {
                    success = true,
```

```
                    message = "Successfully exported " + selectedFormat + " file: " +
fileInfo.Name,
                    exportedFile = exportFilePath,
                    fileName = fileInfo.Name,
                    fileFormat = selectedFormat,
                    directory = directory,
                    fileSizeKB = fileInfo.Length / 1024
                };
            }
            else
            {
                return new { success = false, error = selectedFormat + " file was not created at
expected location: " + exportFilePath };
            }
        }
        else
        {
            return new { success = false, error = "Failed to export " + selectedFormat + " file: "
+ errorMessage };
        }
    }
    catch (System.Exception ex)
    {
        Console.WriteLine("[ERROR] Exception occurred: " + ex.Message);
        Console.WriteLine("[ERROR] Stack trace: " + ex.StackTrace);
        return new { success = false, error = "Exception: " + ex.Message };
    }
}

// === Export helper functions remain unchanged ===
private static bool ExportSTL(IModelDoc2 swModel, string filePath, out string
errorMessage) { ... }
private static bool ExportSTEP(IModelDoc2 swModel, string filePath, out string
errorMessage) { ... }
private static bool ExportPDF(SldWorks swApp, IModelDoc2 swModel, string filePath, out
string errorMessage) { ... }
private static bool ExportIGES(IModelDoc2 swModel, string filePath, out string
errorMessage) { ... }
private static bool ExportParasolid(IModelDoc2 swModel, string filePath, out string
errorMessage) { ... }
}
```

Copie de Copie de Copie de Copie de Copie de Copie de Copi…

Automation title :

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

Copie de Copie de Copie de Copie de Copie de Copie de … (1)

Automation title :

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

Copie de Copie de Copie de Copie de Copie de Copie de … (2)

Automation title :

Here is the download link for the tool where you can create/remix and run those automations with no coding knowledge : https://mecagent-api-production-4ff2.up.railway.app/

Here is the discord for the feedbacks : https://discord.gg/4pCT8FdQXR

Here is the code :

Onglet 21