SolidWorks API Project Manager: Application Documentation

CCET 4610 Spring 2023

Developed by: Weston Shakespear

Taskpane.cs

```
1 reference
public void addDocEntry(string name, string version, int status)
{

    versions.Add(name, version);
    reference.Add(name, this.currentRow);
    auto.Add(name, true);

    backRef[this.currentRow] = name;


    if (this.currentRow < 24)
    {
        this.filenameButtons[this.currentRow].Text = name;
        this.versionLabels[this.currentRow].Text = version;
        this.versionLabels[this.currentRow].BackColor = this.orangeColor;
    }

    this.updateLabel(this.currentRow, status);
    this.updateAutoButtons();
    this.currentRow++;
}
```

Once a new file has been opened this method is called by the main application and is given the opened file information. This information is used to update the Open File table and all the data is kept in a couple of dictionaries called versions, reference, and auto.

```
public void updateDocEntry(string name, string version, int status)
{
    this.versions[name] = version;
    this.versionLabels[this.reference[name]].Text = version;

    int index = this.reference[name];

    this.updateLabel(index, status);
}

2 references
public void updateLabel(int index, int status)
{
    this.versionLabels[index].BackColor = status switch
    {
        0 => this.greenColor,
        1 => this.orangeColor,
        _ => this.redColor
    };
}

1 reference
public void removeDocEntry(string name)
{
    int i = this.reference[name];
    this.filenameButtons[i].Text = "";
    this.versionLabels[i].Text = "";
    this.versionLabels[i].BackColor = this.backColor;
    this.autoButtons[i].BackColor = this.backColor;

    versions.Remove(name);
    reference.Remove(name);
    auto.Remove(name);
}
```

These methods are called by the main application in order to update or remove the document entries from the Open File table. The method updateDocEntry takes the status and version of the file and modifies the cell colors and text to match.

```
1 reference
344     public void FileButtonClicked(object sender, EventArgs args)
345     {
346         if (sender != null)
347         {
348             Button? btn = sender as Button;
349             if (btn != null)
350             {
351                 string name = this.backRef[Int32.Parse(btn.Name)];
352                 this.NameClicked(sender, args, name);
353             }
354         }
355
356
357
358     }
359
                1 reference
360     public void AutoButtonClicked(object sender, EventArgs args)
361     {
362         if (sender != null)
363         {
364             Button? btn = sender as Button;
365             if (btn != null)
366             {
367                 string name = this.backRef[Int32.Parse(btn.Name)];
368
369                 this.auto[name] = !this.auto[name];
370                 this.AutoSyncChanged(sender, args, name, this.auto[name]);
371
372                 this.updateAutoButtons();
373
374
375             }
376         }
377     }
378
                2 references
379     private void updateAutoButtons()
380     {
381         foreach (KeyValuePair<string, bool> autoCell in this.auto)
382         {
383             Color back = this.greenColor;
384
385             if (!autoCell.Value)
386             {
387                 back = this.redColor;
388             }
389             int index = this.reference[autoCell.Key];
390             this.autoButtons[index].BackColor = back;
391         }
392     }
```

These are the event handlers for clicking the filename or auto-sync button for each entry in the Open File table.

Settings.cs

```
25  public Settings(string address, string user, string path, string pid)
26  {
27      InitializeComponent();
28      this.address = address;
29      this.user = user;
30      this.path = path;
31      this.pid = pid;
32
33  }
34
    2 references
35  private void updateValues()
36  {
37      this.address = addressTextBox.Text;
38      this.user = userTextBox.Text;
39      this.path = pathTextBox.Text;
40  }
41
    1 reference
42  private void applyButton_Click(object sender, EventArgs e)
43  {
44      this.updateValues();
45      this.Close();
46  }
47
    1 reference
48  private void pathTextBox_Click(object sender, EventArgs e)
49  {
50      FolderBrowserDialog rootFolder = new FolderBrowserDialog();
51      DialogResult result = rootFolder.ShowDialog();
52
53      if (result == DialogResult.OK)
54      {
55          pathTextBox.Text = rootFolder.SelectedPath;
56      }
57  }
58
    1 reference
59  private void loadButton_Click(object sender, EventArgs e)
60  {
61      OpenFileDialog uploadFileDialog = new OpenFileDialog();
62      DialogResult result = uploadFileDialog.ShowDialog();
63
64      if (result == DialogResult.OK)
65      {
66          dynamic o1 = JObject.Parse(File.ReadAllText(uploadFileDialog.FileName));
67          string address = o1.address;
68
69          if (address != null)
70          {
71              this.addressTextBox.Text = address;
72          }
73
74          string user = o1.user;
75          if (user != null)
76          {
77              this.userTextBox.Text = user;
78          }
79
80          string root = o1.root;
81          if (root != null)
82          {
83              this.pathTextBox.Text = root;
84          }
85      }
86
87      this.updateValues();
88      //this.Close();
89  }
```

This is all of the code for the settings dialog box. The initializer on line 25 takes the existing setting information if there is any and sets the texts boxes to that upon opening.

Line 42 contains the event handler for the apply button, this takes all the data from the text boxes and gives it back to the main application. The pathTextBox_Click method on line 48 triggers the open folder dialog to show when you click it.

Form1.cs

```
1 reference
127     public Form1()
128     {
129
130         InitializeComponent();
131
132         this.FormClosing += new FormClosingEventHandler(this.ApplicationClose);
133
134         // init any colors
135         this.sldConnectLabel.BackColor = this.redColor;
136         this.cloudConnectLabel.BackColor = this.redColor;
137
138         this.solidConnectButton.ForeColor = this.greenColor;
139         this.apiConnectButton.ForeColor = this.greenColor;
140
141         // find and set location of executable
142         var location = System.Reflection.Assembly.GetExecutingAssembly().Location;
143         if (location != null)
144         {
145             var path = Path.GetDirectoryName(location);
146             if (path != null)
147             {
148                 this.executableDirectory = path;
149             }
150
151         }
152
153         if (this.executableDirectory == null)
154         {
155             MessageBox.Show("Error location executable directory, exiting");
156             System.Windows.Forms.Application.Exit();
157         }
158
159
160         // try to load global settings, if not found prompt for file
161         this.initSettings(false);
162
163
164         // try to connect to server with settings in config
165         this.initAPIConnect(this.url, this.user, this.localHead);
166         this.initSolidConnect();
167         //this.update();
168
169
170         // create timers and start
171         this.checkTrackedTimer.Tick += new EventHandler(TimerTrackedEventProcessor);
172         this.checkTrackedTimer.Interval = 1000;
173         this.checkTrackedTimer.Start();
174
175         this.checkDocTimer.Tick += new EventHandler(TimerCheckDocEventProcessor);
176         this.checkDocTimer.Interval = 1000;
177
178     }
```

The initialization method for the main form first registers an event handler to take care of the taskpane destruction on line 132. Then the connection labels are intialized to their default colors on lines 135-139. After that the System.Reflection.Assembly.GetExecutingAssembly() method is used to find the directory, the executable resides in.

After some checks on the executable directory validity the settings are initilized from the file on line 161 and are used to try and initiate the Server connection and Solidworks API connection on line 165 and 166.

Finally the timers for the document saver and document checker are started.

```
5 references
private void update()
{
    if (!this.isAPIConnected())
    {
        return;
    }

    var treeNew = this.lfm.updateCloudTree();
    if (treeNew != null)
    {
        this.tree = treeNew;

        this.updateProjectTree();
        this.updateFileTree();
    }


    this.updateLocal();

    this.refreshPreview();
}
```

Whenever an api action finishes successfully the update method will be called. This in turn gets the new project and file list from the server and then updates the Project and File trees with this information.

```
1 reference
236     private void saveQueueProcessing()
237     {
238         List<string> complete = new List<string>();
239
240         foreach (string item in this.saveQueue)
241         {
242             this.fileSaved(item);
243             // pop item
244             complete.Add(item);
245         }
246
247         foreach (string item in complete)
248         {
249             this.saveQueue.Remove(item);
250         }
251     }
252
    1 reference
253     private void openQueueProcessing()
254     {
255         List<string> exist = new List<string>();
256
257         foreach (KeyValuePair<string, bool> item in this.openQueue)
258         {
259             if (Path.Exists(item.Key))
260             {
261                 exist.Add(item.Key);
262                 this.openFile(item.Key, item.Value);
263             }
264
265         }
266
267         foreach (string item in exist)
268         {
269             this.openQueue.Remove(item);
270         }
271     }
```

The save queue and open queue work almost the same but are for opening a file or saving a file respectively. The saveQueueProcessing method runs every time the checkFile timer starts and checks for files that need to be uploaded to the server. After a successful upload the key is removed from the queue.

The open file queue works the same way and watches for what files are currently open.

```
1 reference
private void closeCheckProcessing()
{
    if (this.sld == null)
    {
        return;
    }
    var value = this.sld.EnumDocuments2();

    List<string> openDocNames = new List<string>();

    if (value != null)
    {
        ModelDoc2 model;

        while (1 == 1)
        {
            int fetched = 0;
            value.Next(1, out model, ref fetched);

            if (model == null)
            {
                break;
            }
            try
            {
                openDocNames.Add(Path.GetFileName(model.GetPathName()));
            }
            catch (System.Runtime.InteropServices.COMException e)
            {

            }


        }
    }

    // loop and remove
    // if value was null this will be all
    foreach (KeyValuePair<string, string> trackedFile in this.trackedFiles)
    {
        string name = trackedFile.Key;

        if (!openDocNames.Contains(name))
        {
            this.removeClosedDocument(name);
        }
    }
}
```

This function is periodically called by the timer and is responsible for tracking the open files and checking for closed files. These are compared to a list of tracked files and is the main way the application keeps track of the file open states and already opened files.

```
1 reference
private void fileSaved(string name)
{
    //name is filename
    if (!this.isAPIConnected())
    {
        return;
    }

    if (this.autoSync.ContainsKey(name))
    {
        if (this.autoSync[name] == false)
        {
            this.taskPaneView.updateDocEntry(name, this.lfm.getLocalFileVersion(name), 1);
            return;
        } else
        {
            string fullPath = this.lfm.getFullPathFromName(name);
            bool res = this.createPreview(fullPath, fullPath + ".BMP", false);


            string dependents = this.getFileDependents(name);
            Debug.WriteLine(dependents);

            if (res)
            {
                this.lfm.uploadFile(name, dependents);
            }

            this.update();
            //status 0 good
            //status 1 new
            //status 2 old
            this.taskPaneView.updateDocEntry(name, this.lfm.getLocalFileVersion(name), 0);
        }

    }

}
```

For each file in the saveQueue this method is called which creates the file preview, lists the dependents, and gives this metadata to the FileManager in order to upload it to the server.

```
1 reference
private void openFile(string openPath, bool ro)
{
    string ext = Path.GetExtension(openPath);
    string modelname = Path.GetFileNameWithoutExtension(openPath);
    string filename = Path.GetFileName(openPath);

    this.docM.openDoc(openPath, ro);
    ModelDoc2 newModel = this.docM.getModelFromName(modelname);

    bool result = ext switch
    {
        ".SLDPRT" => this.attachPartEvents(newModel),
        ".SLDASM" => this.attachAssemblyEvents(newModel),
        ".SLDDRW" => this.attachDrawingEvents(newModel),
        _ => false
    };


    int? status = this.lfm.getFileStatus(filename);
    if (status.HasValue)
    {
        this.addTrackedFile(filename, (int)status);
        this.createEvent();
        Debug.WriteLine(this.docM.getDependents(modelname));
    }
}
```

Using the SWLib and FileManager library I developed this is all the code needed to open a file and attach the correct event handlers based on the file extension.

```
2 references
private void refreshPreview()
{
    string urlData = this.url;
    urlData += "preview/" + this.currentSelectedProject + "/" + this.currentFile;

    Debug.WriteLine(urlData);
    try
    {
        this.previewPictureBox.Load(urlData);
        this.previewPictureBox.Update();
    }
    catch (System.Net.WebException)
    {
        Debug.WriteLine("404");
    }
    catch (System.ArgumentException)
    {
        Debug.WriteLine("fatal");
    }
}
```

This method takes the selected project and selected file and builds a URL out of that which is then loaded into the previewPictureBox. This allows for the previews to be viewed without actually downloading the file.

```
2 references
private void updateProjectTree()
{
    projectTreeView.BeginUpdate();
    projectTreeView.Nodes.Clear();

    foreach (KeyValuePair<string, List<string>> entry in this.tree)
    {
        TreeNode node = new TreeNode(entry.Key);
        node.ForeColor = this.greenColor;
        if (entry.Key == this.currentSelectedProject)
        {
            node.NodeFont = new Font("Calibri", 16, FontStyle.Bold);
        }
        else
        {
            node.NodeFont = new Font("Calibri", 16);
        }

        projectTreeView.Nodes.Add(node);
    }

    projectTreeView.EndUpdate();
}
```

Once the list of server projects and files has been updated this function creates the nodes on the Project Tree and modifies the font depending on the active project.

```
2 references
private void updateFileTree()
{
    if (this.currentSelectedProject != null)
    {
        if (tree.ContainsKey(this.currentSelectedProject))
        {
            fileTreeView.BeginUpdate();
            fileTreeView.Nodes.Clear();

            Dictionary<string, string> nodeTypes = new Dictionary<string, string>();
            nodeTypes.Add("Parts", ".SLDPRT");
            nodeTypes.Add("Assemblies", ".SLDASM");
            nodeTypes.Add("Drawings", ".SLDDRW");

            Dictionary<string, TreeNode> topNodes = new Dictionary<string, TreeNode>();

            foreach(KeyValuePair<string, string> defNode in nodeTypes)
            {
                topNodes.Add(defNode.Value, new TreeNode(defNode.Key));

                this.fileTreeView.Nodes.Add(topNodes[defNode.Value]);
            }


            foreach (var file in tree[this.currentSelectedProject])
            {
                TreeNode node = new TreeNode(file);

                int? st = this.lfm.getFileStatus(file);
                if (st != null)
                {
                    node.ForeColor = status[(int)st];
                }

                string ext = Path.GetExtension(file);
                topNodes[ext].Nodes.Add(node);

            }

            fileTreeView.EndUpdate();
        }
    }

}
```

When a project is selected the file tree for that project will update with the child files. This method takes care of updating those from the Dicitionary supplied by the server. The files are sorted based on type and then added with the font and color reflecting the file status.

```
1 reference
874     private void createEvent()
875     {
876         this.sld = (SldWorks)this.app;
877         this.sld.ActiveDocChangeNotify += this.SldWorks_ActiveDocChangeNotify;
878         this.sld.DocumentLoadNotify2 += this.SldWorks_DocumentLoadNotify;
879         this.sld.FileNewNotify2 += this.SldWorks_FileNewNotify;
880         //this.sld.FileCloseNotify += this.SldWorks_FileCloseNotify;
881     }
882
1 reference
883     private int SldWorks_ActiveDocChangeNotify()
884     {
885         log("Event: SldWorks: Document Changed");
886
887         if (this.checkDocFlag == false)
888         {
889             this.checkDocFlag = true;
890         }
891
892
893         return 1;
894     }
1 reference
895     private int SldWorks_DocumentLoadNotify(string docTitle, string docPath)
896     {
897         log("Event: SldWorks: " + docTitle + " Loaded");
898         return 1;
899     }
1 reference
900     private int SldWorks_FileNewNotify(object NewDoc, int DocType, string TemplateName)
901     {
902         log("Event: SldWorks: New " + TemplateName + " created");
903         return 1;
904     }
```

This method attaches the event handlers to the main Solidworks instance once the connection is successful. These handlers will keep track of when a file opens, closes, and the already open files in future updates.

```
1 reference
1301    void createTaskPane(ISldWorks app)
1302    {
1303        string[] a = { this.executableDirectory, "small.ico" };
1304        string bitmap = Path.Combine(a);
1305        string toolTip = "API Manager";
1306        string ctrlName = "API.Manager";
1307        string ctrlLicKey = "";
1308
1309        swTaskPane = (TaskpaneView)app.CreateTaskpaneView2(bitmap, toolTip);
1310
1311        this.taskPaneView = new Taskpane();
1312        this.taskPaneView.exportSTLButton.Click += this.exportSTLButton;
1313        this.taskPaneView.exportParasolidButton.Click += this.exportParasolidButton;
1314        this.taskPaneView.exportSTEPButton.Click += this.exportSTEPButton;
1315        this.taskPaneView.exportDXFButton.Click += this.exportDXFButton;
1316
1317
1318        this.taskPaneView.AutoSyncChanged += this.taskAutoSyncChanged;
1319        this.taskPaneView.NameClicked += this.taskNameClicked;
1320        swTaskPane.DisplayWindowFromHandlex64(this.taskPaneView.getHandle());
1321
1322        swTaskPane.TaskPaneActivateNotify += swTaskPane_TaskPaneActivateNotify;
1323        swTaskPane.TaskPaneDestroyNotify += swTaskPane_TaskPaneDestroyNotify;
1324        swTaskPane.TaskPaneToolbarButtonClicked += swTaskPane_TaskPaneToolbarButtonClicked;
1325
1326    }
```

The createTaskPane method is used to setup the form and hand it off to the Solidworks instance. First the icon and taskpane name are used to create a new taskpane view on lines 1303-1309. Then a new Taskpane form is created and then button event handlers are connected to it. After this the rest of the taskpane events are connected.

```
1 reference
int swTaskPane_TaskPaneActivateNotify()
{
    if (swTaskPane.GetButtonState(0) == false)
    {
        for (buttonIdx = 0; buttonIdx <= 20; buttonIdx++)
        {
            swTaskPane.SetButtonState(buttonIdx, true);
        }
    }
    else
    {
        for (buttonIdx = 0; buttonIdx <= 20; buttonIdx++)
        {
            swTaskPane.SetButtonState(buttonIdx, false);
        }
    }
    return 0;
}

1 reference
int swTaskPane_TaskPaneDestroyNotify()
{
    return 1;
}

1 reference
int swTaskPane_TaskPaneToolbarButtonClicked(int ButtonIndex)
{
    switch ((ButtonIndex + 1))
    {
        case 1:
            log("Task back button: clicked");
            break;
        case 2:
            log("Task next button: clicked");
            break;
        case 3:
            log("Task cancel button: clicked");
            break;
        case 4:
            log("Task ok button: clicked");
            break;
    }
    return 1;
}
```

These methods are for future use, they keep track of the taskpane view activity and allow you to also have the default check mark buttons on the taskpane.

```
// TestForm Taskpane control events
1 reference
private void exportSTLButton(object sender, EventArgs e)
{
    this.docM.exportActiveDoc("-" + this.getDateTimeString(), "STL");
}
1 reference
private void exportParasolidButton(object sender, EventArgs e)
{
    this.docM.exportActiveDoc("-" + this.getDateTimeString(), "x_t");
}
1 reference
private void exportSTEPButton(object sender, EventArgs e)
{
    this.docM.exportActiveDoc("-" + this.getDateTimeString(), "STEP");
}
1 reference
private void exportDXFButton(object sender, EventArgs e)
{
    this.docM.exportDWG("-" + this.getDateTimeString());
}
```

When triggered by the respective taskpane buttons these event handlers will call the needed SWLib method to export the active file.

```
1 reference
1405    private int PartDocFileSavePostNotify(string name)
1406    {
1407        this.saveQueue.Add(Path.GetFileNameWithoutExtension(name) + ".SLDPRT");
1408        //return 1 if you don't want to save
1409        return 0;
1410    }
1 reference
1411    private int AssemblyDocFileSavePostNotify(string name)
1412    {
1413        this.saveQueue.Add(Path.GetFileNameWithoutExtension(name) + ".SLDASM");
1414        return 0;
1415    }
1 reference
1416    private int DrawingDocFileSavePostNotify(string name)
1417    {
1418        this.saveQueue.Add(Path.GetFileNameWithoutExtension(name) + ".SLDDRW");
1419        return 0;
1420    }
1421
1422
2 references
1423    private bool attachPartEvents(ModelDoc2 model)
1424    {
1425        PartDoc doc = (PartDoc)model;
1426        doc.FileSaveNotify += this.PartDocFileSavePostNotify;
1427        return true;
1428    }
1429
2 references
1430    private bool attachAssemblyEvents(ModelDoc2 model)
1431    {
1432        AssemblyDoc doc = (AssemblyDoc)model;
1433        doc.FileSaveNotify += this.AssemblyDocFileSavePostNotify;
1434
1435        return true;
1436    }
1437
2 references
1438    private bool attachDrawingEvents(ModelDoc2 model)
1439    {
1440        DrawingDoc doc = (DrawingDoc)model;
1441        doc.FileSaveNotify += this.DrawingDocFileSavePostNotify;
1442        return true;
1443    }
```

The first three methods PartDocFileSavePostNotify, AssemblyDocSavePostNotify, and DrawingDocFileSavePostNotify are called respectively for a part, assembly or drawing document once it has been opened by the application.

```
2 references
private void initSettings(bool prompt)
{
    string internalSettings = "";
    bool settingsLoaded = false;

    if (prompt)
    {
        OpenFileDialog dialog = new OpenFileDialog();
        DialogResult res = new DialogResult();

        res = dialog.ShowDialog();

        if (res == DialogResult.OK)
        {
            internalSettings = dialog.FileName;
        }
    }
    else
    {
        string[] subs = { this.executableDirectory, this.settingsFilename };
        internalSettings = Path.Combine(subs);
    }


    try
    {
        dynamic o1 = JObject.Parse(File.ReadAllText(internalSettings));

        var address = o1.address;
        var user = o1.user;
        var root = o1.root;
        var temp = o1.template;
        var pid = o1.pid;

        bool result = (address != null) && (user != null) &&
                      (root != null) && (pid != null) && (temp != null);

        if (result)
        {
            this.url = address;
            this.user = user;
            this.localHead = root + "\\";
            this.templateDir = temp + "\\";
            this.pid = pid;

            settingsLoaded = true;
            this.lfm = new LocalFileManage(this.localHead, this.templateDir);

        }
        else
        {
            MessageBox.Show("Settings file is/has become corrupt");
        }
    }
    catch (System.IO.FileNotFoundException)
    {
        MessageBox.Show("Settings file not found");
        settingsLoaded = false;
    }

    if (!settingsLoaded)
    {
        this.initSettings(true);
    }
}
```

The initSettings method contains the logic for trying to locate the settings file. Upon finding an error in the supplied configuration file or not finding a file at all the method will call iteself again this time prompting for the settings file location.

Once the file is found the JObject.Parse method is used to convert the text into a variable the code can work with, then all the settings values are parsed out as long as they aren't empty or null (don't exist).