Application Documentation: AI-Generated Summaries

Powered by Llama 3 via Ollama

Generation Date: 2025-11-05

Code File: AssemblyInfo.cs

Summary:

****ThemeInfo:** The ThemeInfo class provides information about the theme resources used by an application. This class allows developers to specify where theme-specific resource dictionaries are located and where generic resource dictionaries can be found. Public Interface: * None (this is a class that provides metadata, not a function with inputs or outputs) Purpose: The ThemeInfo class helps the .NET Framework locate theme-related resources, such as styles, templates, and other visual elements. By specifying the locations of these resources, developers can ensure that their application uses the correct themes and styling for different environments and cultures.**

Code File: DocumentController.cs

Summary:

Here is the technical summary: **DocumentController: The DocumentController class synchronizes Documents with UI elements that represent these Documents. It provides a way to react to changes in the Documents collection and the currently active Document. **Public Interface:** * The class takes an IFileService instance as input. * It exposes three abstract methods: + OnDocumentAdded(IDocument document): Called when a new Document is added. + OnDocumentRemoved(IDocument document): Called when a Document is removed. + OnActiveDocumentChanged(IDocument? activeDocument): Called when the currently active Document changes. **Purpose:** The DocumentController class enables your application to respond to changes in the Documents collection and the currently active Document, allowing you to update UI elements accordingly.**

Code File: FileController.cs

Summary:

This is a C# class that appears to be an implementation of the Singleton pattern, specifically designed for a technical writer role in a software company. The class has several methods and

properties related to synchronizing data with a UFM (Unified File Management) server, as well as loading and initializing various types of data from a database. Here are some key features and observations: 1. **Singleton pattern: The class is designed using the Singleton pattern, which ensures that only one instance of the class exists throughout the application. 2. **UFM synchronization**: The class has several methods related to synchronizing data with a UFM server, including `SynchronizeDbLookupDefinitions`, `SynchronizeDbStateDefinitions`, `SynchronizeDbSizeTypes`, and others. 3. **Database loading**: The class also has methods for loading various types of data from a database, such as size ranges, units, workshop times, and custom colors. 4. **Event handling**: The class has an event handler method called `OnAppObjectConnectionModeChanged` that is triggered when the application's connection mode changes. 5. **Dispose pattern**: The class implements the Dispose pattern to ensure proper cleanup of resources when the object is no longer needed. Some potential improvements or considerations: 1. **Code organization**: The class has a lot of methods and properties, which can make it difficult to read and maintain. Consider breaking down the code into smaller, more focused classes or modules. 2. **Error handling**: The class does not appear to have any explicit error handling mechanisms. Consider adding try-catch blocks or other error handling strategies to handle potential exceptions. 3. **Code comments**: While there are some comments in the code, they could be more detailed and descriptive to help others understand the purpose and behavior of each method and property. 4. **Performance optimization**: The class has several methods that may involve database queries or network requests. Consider optimizing these methods for performance, such as by using caching or asynchronous programming. Overall, this is a complex class with many responsibilities, but it appears to be well-organized and easy to understand.**

Code File: IModuleController.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **IModuleController: The IModuleController interface provides a set of methods for managing and controlling a module in an application. The main functions are: * **Initialize:** Initializes the module controller. * **Run:** Runs the module controller. * **Shutdown:** Shuts down the module controller. * **SaveSettings:** Saves settings. These methods allow developers to create modules that can be easily integrated into the application, providing a standardized way to manage and control module behavior.**

Code File: ModuleController.cs

Summary:

****ModuleController:** The ModuleController is responsible for managing the Procost application's main window and its various components. It provides a public interface for interacting with the application, including commands for searching, displaying Intelli menus, exiting, and releasing all instances. **Purpose:** The primary purpose of the ModuleController is to initialize and manage the Procost application's main window, including its views, models, and services. It also handles various events and commands related to the application's functionality. **Public Interface:** * `Initialize()`: Initializes the module controller by setting up various components and services. * `Run()`: Runs the application's main window and initializes its views and models. * `Shutdown()`: Shuts down the application, saving settings and releasing resources. * `SaveSettings()`: Saves the application's settings for all domains. **Commands:** ***

`SearchMenuCommandExec()`: Toggles the search menu on or off. * `IntelliMenuCommandExec()`: Displays Intelli menus. * `ExitCommandExec()`: Closes the main window and releases resources. * `ExitAndReleaseAllCommandExec()`: Closes all instances of the application and releases resources. **Events: * `ShellViewModelClosing(object sender, CancelEventArgs e)`: Handles the closing event of the shell view model, saving settings and releasing resources.**

Code File: OptionsWindowController.cs

Summary:

****OptionsWindowController:** The OptionsWindowController is responsible for managing the behavior of the Options Window in the Procam Procost application. This controller provides a public interface to interact with the Options Window, allowing users to customize various settings and options. **Public Interface:** * The controller exposes two commands: + `SelectedItemChangedCommand`: Triggered when the user selects an option from the list. + `OkCommand`: Triggered when the user clicks the OK button to apply changes. * The controller also provides a property, `CurrentOptionViewModel`, which reflects the currently selected option. **Purpose:** The OptionsWindowController is designed to facilitate customization of various settings and options in the Procam Procost application. It allows users to navigate through different sections (e.g., Customize Ribbon, Quick Access Toolbar, etc.) and apply changes as needed. The controller ensures that any changes made are persisted and reflected in the application's behavior. **Key Features:** * Manages the Options Window's state and behavior * Provides a public interface for interacting with the Options Window * Supports customization of various settings and options * Ensures persistence of changes made by the user**

Code File: PagingController.cs

Summary:

****PagingController:** The PagingController class helps coordinate paged access to a data store. It provides commands for navigating through pages of data, including going to the first page, last page, next page, and previous page. Public Interface: * Initializes with item count, page size, and an IAppService instance. * Provides four commands: GoFirstPageCommand, GoLastPageCommand, GoNextPageCommand, and GoPreviousPageCommand. * Exposes properties for item count, page size, current page, and page count. * Raises events when the current page changes. Purpose: The PagingController class enables efficient navigation through large datasets by dividing them into manageable pages. It provides a set of commands to move between pages, allowing users to easily access specific parts of the data.**

Code File: ProcostDocumentController.cs

Summary:

Here is the technical summary of the C# code: **ProcostDocumentController: The ProcostDocumentController class is responsible for synchronizing Procost Documents with StyleDocumentViewModels. It provides a public interface to manage documents, including**

adding, removing, and activating documents. **Public Interface: * Adds a document: The controller creates a new StyleDocumentViewModel instance and associates it with the added document. * Removes a document: The controller disposes of the removed document's associated StyleDocumentViewModel instance. * Activates a document: The controller updates the application title to reflect the active document's file name, and sets the corresponding StyleDocumentViewModel instance as open, selected, and active. **Purpose:** The ProcostDocumentController class enables the synchronization of Procost Documents with StyleDocumentViewModels, allowing for efficient management of documents in the application.**

Code File: Document.cs

Summary:

Here is the technical summary: **Document: The Document class provides a foundation for creating documents in your application. It allows you to track file name and modification status. **Public Interface:** * **FileName**: Retrieves or sets the file name associated with the document. * **Modified**: Retrieves or sets a flag indicating whether the document has been modified. * **Gid**: An abstract property that returns a unique identifier for the document.**

Code File: IDocument.cs

Summary:

****IDocument:** The IDocument interface provides a way to manage and track document-related information. It allows you to specify the file name, indicate whether the document has been modified, and retrieve a unique identifier (Gid) for the document. Public Interface: * FileName: Get or set the file name of the document. * Modified: Get whether the document has been modified. * Gid: Get the unique identifier (Guid) for the document.**

Code File: MainStyleDocument.cs

Summary:

Here is the technical summary: **MainStyleDocument: The MainStyleDocument class provides a way to manage and interact with style data in an application. It takes an IStyle object as input, which represents the main style document. **Public Interface:** * The class exposes several properties that allow access to the underlying style data: + UnderworkStyle: The main style document. + Style: A read-only property that returns the working style. + Name: The name of the style document. + Gid: The globally unique identifier for the style document. + Modified: A flag indicating whether the style document has been modified. **Methods:** * AcceptAllChanges(): Accepts all changes made to the underlying style data. * RevertAllChanges(): Reverts all changes made to the underlying style data. * ThrowAllChangesBeforeRemove(): Throws an exception if there are any pending changes before removing the style document. * IsValidationError(): Checks whether there is a validation error in the underlying style data. **Disposal:** The class implements the IDisposable interface, which means it needs to be properly disposed of when no longer needed. This ensures that any resources used by the class are released and cleaned up correctly.**

Code File: ClipboardHelper.cs

Summary:

Here is the technical summary: **ClipboardHelper: The ClipboardHelper class provides functionality for copying and pasting style data and structures between different parts of an application. It allows users to copy a selection of styles and then paste them into another area, maintaining their hierarchical structure. **Public Interface:** * **CopyStyleDataHandle**: Copies the specified styles into the clipboard. * **CanPasteStyleDataHandle**: Determines whether the clipboard contains valid style data that can be pasted. * **PasteStyleDataHandle**: Pastes the copied style data into the target area, duplicating parts and adding patterns as needed. * **CanPasteStyleStructureHandle**: Checks if the clipboard contains a single style that can be pasted into multiple styles. * **PasteStyleStructureHandle**: Pastes the copied style structure into the target area, copying the hierarchy of styles.**

Code File: CommonHelper.cs

Summary:

Here is the summary of the `CommonHelper` class: **GetLookupLevel: Returns a string indicating the type of lookup level (style, variant, material, or part) based on the input type. **GetSelectedDbLookupItems**: Returns a list of selected database lookup items for a given type (style, variant, material, or part). **GetExcludedLookupDefinitions**: Returns a list of excluded lookup definitions for a given type (style, variant, material, or part). **GetObligatoryFields**: Returns a list of obligatory fields for a given style data. **GetPredefinedFields**: Returns a dictionary of predefined fields for a given style data. **GetTypesList**: Asynchronously returns a list of material types from the database and lookup definitions. **GetTypesListExt**: Asynchronously returns a list of unique material types by combining results from the database and lookup definitions. **CostCategoryList**: Returns an array of cost category names. **GetAllPossibleGuids**: Recursively returns a list of possible GUIDs for a given style data, including its variants, materials, and parts. **MaterialAlreadyExists**: Checks if a material already exists in a variant's materials. **AreStyleDataSiblings**: Determines if multiple style data items are siblings (i.e., have the same parent). **GetCommonSourceName**: Returns the common source name for a list of style data items that are siblings. **GetCommonParentStyle**: Returns the common parent style for a list of sibling style data items. **SupportedFileTypes**: Returns a list of supported file types, including SVG, SEM, and DXF files. **CheckUfmState**: Checks if an UFMX service result is successful or not. If not, logs a warning message. **LogUfmState**: Logs the state of UFMX service synchronization with the local database. **KnownColors**: Returns a dictionary of known colors, mapping color names to MediaColor values.**

Code File: DocumentViewsHelper.cs

Summary:

Here is the technical summary of the `DocumentViewsHelper` class: **GetOrCreateStyleDocumentFromStyle: Gets or creates a `MainStyleDocument` from an `IStyle` object. If a document with the same GID already exists, it returns that document; otherwise, it creates a new one and adds it to the file service.**

GetOrCreateSubStyleDocumentViewModel: Gets or creates a `SubStyleDocumentViewModel` for a given `StyleDocumentViewModel`, `IStyle`, and boolean flag. If a sub-document with the same name already exists, it returns that document; otherwise, it creates a new one. **GetStyleFromBaseRecord:** Retrieves an `IStyle` object from a base record (e.g., `IStyleData`, `IVariant`, etc.).
GetStyleDataSubDocumentViewModelFromBaseRecord: Gets a `SubDocumentViewModel` for a given `StyleDocumentViewModel`, `IUnderWorkTreeItem`, and optional document type.
GetPatternsGridSubDocumentViewModelFromBaseRecord: Gets a `SubDocumentViewModel` for a given `StyleDocumentViewModel` and `IStyleData`.
GetPatternsListSubDocumentViewModelFromBaseRecord: Gets a `SubDocumentViewModel` for a given `StyleDocumentViewModel` and `IStyleData`.
GetVariantWizardSubDocumentViewModelFromBaseRecord: Gets a `SubDocumentViewModel` for a given `Style` object.
CreateStyleDataSubDocumentViewModelFromBaseRecord: Creates a `SubDocumentViewModel` from an object (e.g., `IStyle`, `IVariant`, etc.) and optional parameter. The type of sub-document created depends on the object's type.
CreatePatternsGridSubDocumentViewModelFromBaseRecord: Creates a `SubDocumentViewModel` for a given `IStyleData`.
CreatePatternsListSubDocumentViewModelFromBaseRecord: Creates a `SubDocumentViewModel` for a given `IStyleData`.
CreateVariantWizardSubDocumentViewModelFromBaseRecord: Creates a `SubDocumentViewModel` for a given `IStyle`.

Code File: ExportImportHelper.cs

Summary:

Here is the technical summary of the `ExportImportHelper` class:
ExportCompanyDataCommandExec: Exports company data to a ZIP file. This command executes when the user initiates an export operation. The function takes no input parameters and returns a ZIP file containing company settings, reports data files, and database data files.
ImporCompanyDataCommandExec: Imports company data from a ZIP file. This command executes when the user initiates an import operation. The function prompts the user to select a ZIP file and then extracts the contents of the file into a temporary location. It then imports the company settings, reports data files, and database data files.
ExportUserSettingsCommandExec: Exports user settings to a JSON file. This command executes when the user initiates an export operation for their user settings. The function prompts the user to select a save location and then exports the user settings in JSON format.
ImporUserSettingsCommandExec: Imports user settings from a JSON file. This command executes when the user initiates an import operation for their user settings. The function prompts the user to select a JSON file and then imports the user settings into the application.
ExportIntelliCommandsCommandExec: Exports IntelliCommands settings to a JSON file. This command executes when the user initiates an export operation for their IntelliCommands settings. The function saves the IntelliCommands settings in JSON format to a specified location. **ImportIntelliCommandsCommandExec:** Imports IntelliCommands settings from a JSON file. This command executes when the user initiates an import operation for their IntelliCommands settings. The function prompts the user to select a JSON file and then imports the IntelliCommands settings into the application.

Code File: FilterHelper.cs

Summary:

Here is the technical summary of the `FilterHelper` class: **FilterHelper: The `FilterHelper` class provides a set of commands and properties to manage filtering in a grid overview. It allows users to filter columns based on specific operators (e.g., contains, starts with) and apply filters to the grid. **Public Interface:** * The class takes a `BaseGridOverviewViewModel` as input. * It exposes several properties: + `ModelBase`: A reference to the underlying view model. + `CurrentPopupColumnName`, `CurrentFilterText`, `CurrentSelectedFilterOperator`, and `CurrentCaseSensitive`: Properties that track the current filter settings. + `IsPopupOpen`: A flag indicating whether the filter popup is open or not. * It also provides several commands: + `ColumnFilterButtonCommand`: Toggles the filter popup for a specific column. + `ResetAllFiltersCommand`: Resets all filters to their default state. + `ApplyNoFilterCommand`: Applies no filter to a specific column. + `ClearFilterBoxCommand`: Clears the filter text box. + `ApplyFilterCommand`: Applies the current filter settings to the grid. + `PopupCloseCommand`: Closes the filter popup. **Purpose:** The `FilterHelper` class enables users to apply filters to a grid overview, allowing them to customize their view and focus on specific data.**

Code File: MessageHelper.cs

Summary:

Here is the summary of the C# code: **MessageHelper: The **MessageHelper** class provides a set of methods for displaying various types of messages to the user. These messages can be questions, warnings, or error messages. **ShowQuestion:** Displays a question message with a list of names and asks the user if they want to save their changes. **ShowSaveDocumentYesNoMessage:** Displays a yes/no question message with a list of names and asks the user if they want to save their document. **ShowWarning:** Displays a warning message with a list of names, indicating that the user needs to save or discard their changes before continuing. **ShowMessage:** Displays a general message with a list of names. **ShowProcostAlreadyRunningMessage:** Displays an error message indicating that Procost is already running. **ShowInvalidLicenseMessage:** Displays an error message indicating that the license is invalid, along with any additional error information provided. **ShowUnsuccessExportMessage:** Displays an error message indicating that an export operation was unsuccessful, including the file name and reason for failure. **ShowUnsuccessImportMessage:** Displays an error message indicating that an import operation was unsuccessful, including the file name and reason for failure. **ShowSuccessImportAndRestartMessage:** Displays a success message indicating that an import operation was successful, along with instructions to check the log window after restarting the application. **ShowSuccessExportMessage:** Displays a success message indicating that an export operation was successful.**

Code File: NavigationHelper.cs

Summary:

Here is the technical writer's summary of the NavigationHelper class: **Class Name: NavigationHelper **Purpose:** The NavigationHelper class provides methods for navigating through a collection of styles, variants, materials, and parts in a software application. It helps users to open, close, or navigate between related documents. **Methods:** 1.**

NavigationFirstHandle: Opens the first document in the collection. 2. **NavigationLastHandle**: Opens the last document in the collection. 3. **NavigationNextHandle**: Opens the next document in the collection, if available. 4. **NavigationPreviousHandle**: Opens the previous document in the collection, if available. **Properties:** 1. **CanNavigationFirst**: Returns a boolean indicating whether the first document can be opened. 2. **CanNavigationLast**: Returns a boolean indicating whether the last document can be opened. 3. **CanNavigationNext**: Returns a boolean indicating whether the next document can be opened. 4. **CanNavigationPrevious**: Returns a boolean indicating whether the previous document can be opened. **Private Methods:** 1. **OpenCloseDocumentHandle**: Opens or closes a document, depending on the user's settings. 2. **GetIndex**: Finds the index of a specific document in a collection based on its GUID. **Notes:** * The NavigationHelper class uses Reactive Extensions (Rx) to handle asynchronous operations and provide real-time feedback to users. * It relies on other classes, such as StyleOperationsHelper, to perform actual document opening and closing operations. * The class is designed to be flexible and adaptable to different user scenarios and settings.

Code File: PictureHelper.cs

Summary:

Here is the technical summary of the C# code: **PictureHelper:** The PictureHelper class provides a set of utility methods for working with images. The main purpose of this class is to convert SVG streams into image formats, fit background rectangles, and create image sources from files. **SvgStreamToImage:** Converts an SVG stream into a .png image, scaling it to fit within the specified maximum width and height. **FitBackgroundRectangle:** Takes an image and fits it within a rectangular area of the specified size, optionally filling the remaining space with a specified color. **CreateImageSourceFromFile:** Creates a System.Windows.Media.ImageSource from a file path, handling exceptions and displaying error messages if necessary. **ConvertImageSourceToBytes:** Converts a System.Windows.Media.ImageSource into a byte array in the specified format (default is .png).

Code File: ResourcesHelper.cs

Summary:

Here is the technical summary: **ResourcesHelper:** The ResourcesHelper class provides two main functions for extracting and processing embedded resources in a .NET assembly. **ExtractFile:** This function extracts a file from an assembly's resources to a specified output path. It takes three inputs: the assembly, the output path, and the name of the resource to extract. The function writes the extracted file to disk. **ExtractXDocumentAndVersionFromEmbeddedXml:** This function extracts an embedded XML document from an assembly's resources and returns it as an XDocument object. Additionally, it retrieves the version attribute from the XML document and outputs it as a double value. The function takes three inputs: the assembly, the name of the resource to extract, and an output parameter for the last version number.

Code File: StyleOperationsHelper.cs

Summary:

This is a C# code for an expert technical writer for a software company. The goal of this code is to generate various types of documentation, such as user manuals, guides, and tutorials, based on the software's features and functionality. The code defines several methods that can be used to generate different types of documentation: 1. `CalculateStyle`: This method calculates the style of a given set of styles. 2. `CalculateVariant`: This method calculates the variant of a given set of variants. 3. `CalculateMaterial`: This method calculates the material of a given set of materials. 4. `CalculatePart`: This method calculates the part of a given set of parts. The code also includes several other methods that can be used to generate different types of documentation, such as: 1. `OpenDocument`: This method opens a document based on the software's features and functionality. 2. `GetCalculationSources`: This method gets the calculation sources for a given set of styles or variants. 3. `LogCalculationSources`: This method logs the calculation sources for a given set of styles or variants. The code uses various classes and interfaces, such as `IStyle`, `IVariant`, `IMaterial`, and `IPart`, to represent different types of software features and functionality. The code also uses various methods and properties from these classes and interfaces to generate the documentation. Overall, this code provides a framework for generating different types of documentation based on the software's features and functionality.

Code File: UnitOfMethod.cs

Summary:

This is a C# method that populates a list of `UnitOfMethod` objects with various calculation methods and their corresponding unit types, names, sizes, and descriptions. The method iterates through an enum of calculation methods (`Calculation.Method`) and creates a new `UnitOfMethod` object for each method that is not equal to `NONE`. The `UnitOfMethod` class has properties for the method name, unit type, size uplift, and description. Here's a breakdown of the code: 1. The method starts by creating an empty list of strings called `methodsToCheck`, which will store the names of all calculation methods except `NONE`. 2. It then iterates through the enum of calculation methods using `Enum.GetNames()`. This returns an array of string values representing each enum value. 3. The method removes the name of the `NONE` enum value from the list, since we don't want to include it in our report units table. 4. It then iterates through the list of calculation methods and creates a new `UnitOfMethod` object for each method that is not equal to `NONE`. For each method, it sets the properties: * `Method`: The name of the calculation method. * `UnitType`: The unit type (e.g., length, area) corresponding to the method. * `Type`: An integer value representing the type of the method (not used in this example). * `Nameunit`: A string representing the name of the unit (e.g., "cm" for centimeters). * `Sizeuplift`: An integer value representing the size uplift factor (not used in this example). * `Description`: A string describing the calculation method. 5. Finally, the method checks if there are any methods left in the `methodsToCheck` list that were not processed. If so, it throws a `NotImplementedException` with a message indicating which method is missing. The purpose of this method appears to be generating a report units table for a software application that performs various calculations. The table will contain information about each calculation method, including its name, unit type, and description.

Code File: ViewsHelper.cs

Summary:

Here is the technical summary: **ViewsHelper: The ViewsHelper class provides utility methods for working with data grids and sorting. It allows you to convert between SortingDirection and ListSortDirection, as well as retrieve SortDescription from a DataGridColumn. **Public Interface:** * GetSortDescriptionFromDataGridColumn: Takes a DataGridColumn as input and returns a SortDescription. * ConvertSortingDirectionToListSortDirection: Takes a SortingDirection as input and returns the equivalent ListSortDirection (or null if none). * ConvertListSortDirectionToSortingDirection: Takes a ListSortDirection as input and returns the equivalent SortingDirection (or null if none). **Purpose:** The ViewsHelper class simplifies the process of working with data grids and sorting, making it easier to manage and manipulate data in your application.**

Code File: WizardHelper.cs

Summary:

Here is the summary of the `WizardHelper` class: **WizardHelper: This class provides various helper methods for working with wizards in the Procam Procost application. It takes a `SubVariantWizardDocumentViewModel` object as its constructor parameter. **Purpose:** The primary purpose of this class is to facilitate the creation and management of wizard documents, which are used to assign materials to variants. The class provides methods for filtering and sorting data, handling material assignments, and updating the wizard document view. **Public Interface:** * `BindDataViewWithAssigments`: Binds the data view with assignments. * `FilterCollectionView`: Filters a collection view based on a filter text. * `RefreshMaterials`: Refreshes the materials collection based on a filter text. * `WizardCellPrepareForInsertMaterialCommandExec`: Prepares for inserting a material into a wizard cell. * `WizardCellColorPickerCommandExec`: Opens or closes the color picker for a wizard cell. * `HandleMultipleRemoveParts`: Handles removing multiple parts from a wizard document. **Properties:** * `DbMaterials`: A list of materials in the database. * `WizardDataTable`: A data table representing the wizard document. * `SelectedCellsIndexes`: A list of selected cells in the wizard document. **Events:** * `OnCellViewModelAssignmentChanged`: Fires when a cell's material assignment changes.**

Code File: BrowserDataItem.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **ComputerData: Purpose: Holds root My Computer data. Public Interface: - Gets drives (list of drive information) - Gets name ("My Computer") **DriveData:** Purpose: Holds data relating to a drive. Public Interface: - Gets directories (list of directory information) - Gets info (drive information) - Gets name (drive name with type) **BrowserDataItem:** Purpose: Holds data relating to a directory. Public Interface: - Gets directories (list of directory information) - Gets info (directory information) - Gets name (directory name)**

Code File: CompanyDataItem.cs

Summary:

Here is the technical summary: **DataItem: The DataItem class provides a foundation for representing data items in the application. It serves as an abstract base class that can be inherited by specific data item types, such as files and folders. The public interface includes a name property (with both get and set methods) and a selected status indicator. **DataFile:** A DataFile represents a file in the system. It provides information about the file's name, short name, and file info. Users can create instances of DataFile by passing a file path to its constructor. **DataFolder:** A DataFolder represents a folder in the system. It provides information about the folder's name and directory info. Additionally, it maintains a collection of child items (files and folders) through its Items property. Users can create instances of DataFolder by passing a folder path to its constructor.**

Code File: DbLookupItem.cs

Summary:

Here is the technical summary: **DbLookupItemSdi: Provides a way to bind a lookup definition independently of 'underwork' IStyleData. This interface allows you to select or deselect a lookup item, retrieve its selected value and definition, and clone the object. Public Interface: - IsSelected (bool): Indicates whether the lookup item is selected. - SelectedLookupValue (ILookupValue): Retrieves the selected value of the lookup item. - LookupDefinition (ILookupDefinition): Retrieves the definition of the lookup item. - StyleDataSdi (SDI.IStyleDataFields): Retrieves the style data fields associated with the lookup item. **DbLookupItem:** Extends DbLookupItemSdi and provides a way to bind a lookup definition dependent on 'underwork' IStyleData. This class allows you to select or deselect a lookup item, retrieve its selected value and definition, and clone the object. Public Interface: - IsSelected (bool): Indicates whether the lookup item is selected. - SelectedLookupValue (ILookupValue): Retrieves the selected value of the lookup item. - LookupDefinition (ILookupDefinition): Retrieves the definition of the lookup item. - StyleData (IStyleData): Retrieves the style data associated with the lookup item. Note: The classes DbLookupItemSdi and DbLookupItem are used to bind a lookup definition, either independently or dependent on 'underwork' IStyleData.**

Code File: DbOtherFieldItem.cs

Summary:

Here is the technical summary: **DbOtherFieldItem: The DbOtherFieldItem class represents a data item for other fields in an application. It provides a way to store and manage information about these fields, including their name, value, type, selection status, and editing status. Public Interface: - Set or get the field name (OtherFieldName) - Set or get the field value (OtherFieldValue) - Get or set the field type (Type) - Get or set whether the field is selected (IsSelected) - Get or set whether the field is being edited (IsEditing) This class also provides a method to create a clone of itself, which can be useful for creating duplicate items.**

Code File: FilterObj.cs

Summary:

Here is the technical summary: **FilterObj: The FilterObj class provides a way to filter and categorize data in your application. It allows you to set and retrieve two string properties (Title1 and Title2) and a boolean property (IsChecked). The IsChecked property can be used to indicate whether a particular filter is applied or not. This class also overrides the ToString method, which returns a formatted string that includes the type of object, its checked status, and the values of Title1 and Title2.**

Code File: SettingTuple.cs

Summary:

Here is the technical summary: **SettingTuple: The SettingTuple class provides a way to store and manage settings for an application. It allows you to associate a name with a property information object and track whether the setting is selected or not. Public Interface: - Set a name for the setting - Get or set the property information associated with the setting - Track whether the setting is selected or not This class helps applications manage their settings in a flexible and organized way.**

Code File: StyleDataActiveFindObject.cs

Summary:

Here is the technical summary: **StyleDataActiveFindObject: The StyleDataActiveFindObject class provides a way to store and retrieve active style data objects. It has two main properties: `StyleDataEnumActive` and `Ucos`. * `StyleDataEnumActive`: This property allows you to specify an active style data enum value. * `Ucos`: This property is a collection of user column objects that are associated with the active style data. This class enables you to manage and manipulate active style data objects in your application.**

Code File: StyleDataGroup.cs

Summary:

Here is the technical summary: **StyleDataGroup: The StyleDataGroup class provides a way to group and manage style-related data in your application. It allows you to define a group name and associate it with a collection of view names (ViewsNames). Additionally, you can track the currently selected view name (CurrentViewName). Public Interface: - GroupName: A string property that represents the name of the style data group. - ViewsNames: An ObservableCollection of strings that contains the list of available view names. - CurrentViewName: A string property that tracks the currently selected view name. This class is designed to be used in conjunction with other classes and frameworks, such as MvvmBase.Foundation, to provide a robust and flexible way to manage style data in your application.**

Code File: WizardCellViewModel.cs

Summary:

****WizardCellViewModel:** The WizardCellViewModel class is used to manage the behavior and properties of a wizard cell in a Procost application. It provides an interface for users to interact with the cell, including selecting materials, adding parts, and removing parts. **Public Interface:** * The class takes a `WizardHelper` object as a constructor parameter. * It exposes several properties: + `Variant`: Gets or sets the variant material associated with the wizard cell. + `Material`: Gets or sets the material code selected in the combo box. + `Pattern`: Gets or sets the pattern used to determine which parts are available for selection. + `FilterText`: Gets or sets the text displayed in the combo box filter. * It also exposes several commands: + `MouseDoubleClickListBoxCommand`: Invoked when the user double-clicks on a material code in the list box. + `RemovePartCommand`: Invoked when the user removes a part from the wizard cell. + `ColorEnterCommand`: Invoked when the user enters a color for a material. **Behavior:** * When the user selects a new material code, the class updates the combo box filter and raises an `AssignmentChanged` event to notify other parts of the application that the assignment has changed. * When the user adds or removes a part from the wizard cell, the class updates the material code and pattern accordingly. * The class provides methods for handling events such as entering a new material code, removing a part, and adding a material from the database. **Notable Methods:** * `HandleEnteredCode()`: Handles the event when the user enters a new material code. It updates the combo box filter and raises an `AssignmentChanged` event. * `RemovePartCommandExec()`: Handles the event when the user removes a part from the wizard cell. It updates the material code and pattern accordingly. * `ColorEnterCommandExec()`: Handles the event when the user enters a color for a material. It updates the material code and pattern accordingly. **Summary:** The WizardCellViewModel class provides an interface for users to interact with a wizard cell in a Procost application, including selecting materials, adding parts, and removing parts. It exposes several properties and commands that allow other parts of the application to respond to changes in the wizard cell's state.**

Code File: WizardModels.cs

Summary:

Here is the technical summary: **PatternAssigment: The PatternAssigment class manages pattern assignments in an application. It provides a way to associate a pattern with variant groups, which can contain material assignments. Public Interface: - Assigns a pattern (IPattern) to the assignment - Manages variant groups (ObservableCollection), each containing material assignments **VariantGroup:** The VariantGroup class represents a group of variants in an application. It allows you to define a name for the group and manage material assignments within it. Public Interface: - Defines a name for the variant group - Manages material assignments (ObservableCollection)**

Code File: Procam.Procost.Applications.AssemblyInfo.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **Assembly Information: The Assembly Information function provides metadata about the software application, including company name, configuration, copyright information, file version, informational version, product name, title, and version. This information is used to identify the application and its components. Public Interface: None Purpose: To provide essential metadata**

about the software application, allowing for easy identification and management of the application's components.

Code File: DocumentViewsService.cs

Summary:

****DocumentViewsService:** The DocumentViewsService is a service responsible for managing document views and styles collections. It provides a public interface to interact with the application's document views and styles. **Public Interface:** * `GetStyleDocumentViewModel(IDocument document)`: Returns a StyleDocumentViewModel instance associated with the given document. * `CreateOverviewViewModel(string sm, bool isOpen)`: Creates an overview view model for either "Styles" or "Materials", depending on the input parameter `sm`. The `isOpen` parameter determines whether the view is initially open or not. * `RemoveGridsAndFormsSubdocuments(StyleDocumentViewModel styleDocumentViewModel, IList possibleGuids)`: Removes sub-documents (grids and forms) from a StyleDocumentViewModel instance that match the given GUIDs. * `RemoveStyleDataSubdocuments(StyleDocumentViewModel styleDocumentViewModel, IList guids)`: Closes all sub-documents with matching GUIDs in a StyleDocumentViewModel instance. * `CloseSubDocuments(IEnumerable gids)`: Closes all sub-documents with matching GUIDs across all StyleDocumentViewModel instances. * `CloseSubDocument(IViewModel model)`: Closes a specific sub-document (grid or form) associated with the given IViewModel instance. * `GetStylesInDbCount()`: Returns the count of styles in the database. * `GetMaterialsInDbCount()`: Returns the count of materials in the database. * `LoadStylesWithoutChildsAsync(int skipCount, int itemsPerPage, IList sortDescriptions, IEnumerable<(string columnName, Op filterOperator, string filterText, bool caseSensitive)> filters, StyleDataEnum? styleDataEnum)`: Loads styles without child records asynchronously, using the provided parameters. * `LoadMaterialsInStoreAsync(int skipCount, int itemsPerPage, IList sortDescriptions, IEnumerable<(string columnName, Op filterOperator, string filterText, bool caseSensitive)> filters)`: Loads materials in store asynchronously, using the provided parameters. **Events:** The DocumentViewsService raises three events to notify subscribers of changes: * `StylesRepositoryChanged`: Notifies subscribers when styles are updated or changed. * `MaterialsRepositoryChanged`: Notifies subscribers when materials are updated or changed. * `LookupDefinitionsRepositoryChanged`: Notifies subscribers when lookup definitions are updated or changed.**

Code File: ExportImportService.cs

Summary:

Here is the technical summary: **ExportImportService: The ExportImportService class provides a centralized interface for exporting and importing data related to company settings, user settings, styles, and IntelliCommands. This service allows applications to interact with these data types in a standardized way. **Public Interface:** * The service offers various properties that provide access to factories for creating export and import view models. * It also provides several commands for executing export and import operations: + ExportCompanyDataCommand: Exports company data settings. + ImportCompanyDataCommand: Imports company data settings. + ImportStylesCommand: Imports styles settings. + ExportUserSettingsCommand: Exports user settings. + ImportUserSettingsCommand: Imports user settings. + ExportIntelliCommandsCommand:**

Exports IntelliCommands settings. + ImportIntelliCommandsCommand: Imports IntelliCommands settings. **Purpose: The ExportImportService enables applications to manage and synchronize data related to company settings, user settings, styles, and IntelliCommands.**

Code File: ExportPictureService.cs

Summary:

****ExportPictureService:** The ExportPictureService class provides a mechanism for exporting pictures from various data sources. It allows users to specify the export type (SVG, BMP, or other formats), image format, and file path. Public Interface: * `ExportPictureCommand`: A command that initiates the picture export process. * `ExportPictureCommandExec(object param)`: Executes the export command with a given parameter (data source). Functionality: The service exports pictures from various data sources, including styles, materials, parts, and their results. It generates a file name based on the data source and allows users to customize the export settings. When executed, the `ExportPictureCommand` command prompts the user for export settings and then exports the picture to the specified file path in the chosen format (SVG or BMP). The service also updates the user's settings with the exported image information.**

Code File: ExportToExcelReportService.cs

Summary:

This is a C# class for generating Excel reports from data. The class, `ExportToExcelReportService`, has several methods and properties that are used to generate the report. Here's an overview of the class: 1. **Constructor: The constructor initializes some variables and sets up the logger. 2. **Dispose method**: This method is used to clean up any resources (such as temporary files) that were created by the class. 3. **ExportToExcelReport method**: This is the main method that generates the Excel report. It takes an `IXLWorksheet` object as input and uses it to create a new Excel file. 4. **GetBindValue method**: This method is used to get the value of a property based on a binding string. The binding string is used to navigate through the object graph and find the desired property. 5. **RemoveExpandedTemplateArea method**: This method removes any expanded template areas from the worksheet. 6. **ProcessExpandedArea method**: This method processes an expanded area in the worksheet by copying cells and formatting them according to the template. 7. **FillCellsWithValues method**: This method fills cells with values based on a list of cells and their corresponding values. The class also has several properties and methods that are used to generate the report, such as: * `createdTempFilesToRemove`: A list of temporary files that were created during the generation process. * `disposedValue`: A boolean flag that indicates whether the object has been disposed. * `logger`: A logger object that is used to log messages. Overall, this class provides a way to generate Excel reports from data by using a template and filling in the cells with values.**

Code File: ExportToReportService.cs

Summary:

This is a large code block, but I'll try to provide an overview of what it does and highlight some key points. **Technical Writer's Perspective The code appears to be part of a software application that generates reports using a report viewer. The technical writer's role is to document the functionality of this code in a clear and concise manner. **Key Features** 1. **Report Generation**: The code generates reports based on input data and uses a report viewer to display the results. 2. **Pipe Server**: The code uses an anonymous pipe server to communicate with the report viewer process. 3. **Temporary Files and Directories**: The code creates temporary files and directories to store intermediate results and temporary data. 4. **Error Handling**: The code includes error handling mechanisms to catch and handle exceptions that may occur during report generation. **Code Organization** The code is organized into several methods, each with a specific purpose: 1. `ExportToReport_p6` and `ExportToReport_v8`: These methods generate reports using different versions of the report viewer. 2. `FillDataset`: This method populates a dataset with data from an input object. 3. `GenerateXmlDataset`: This method generates an XML file containing the report data. 4. `CallReportViewer`: This method starts the report viewer process and sends it the necessary data to generate the report. **Technical Notes** 1. The code uses .NET 6 features, such as anonymous pipe servers and async/await syntax. 2. The code includes custom classes, such as `XmlTextWriterWithFixedGuidRef`, which are used to handle GUID types in XML writing. 3. The code uses various libraries and frameworks, including ActiproSoftware.* libs. **Documentation Requirements** To effectively document this code, I would recommend the following: 1. **Method-level documentation**: Provide detailed comments for each method, explaining its purpose, input parameters, return values, and any notable logic or exceptions. 2. **Class-level documentation**: Document the classes and their relationships to other parts of the system. 3. **Error handling documentation**: Document the error handling mechanisms used in the code, including any specific exception types and how they are handled. By following these guidelines, you can create a comprehensive technical writer's guide that will help others understand and maintain this codebase.**

Code File: ExportToReport_p6.cs

Summary:

A technical writer's dream! This is a C# class that generates data for a software application. It appears to be responsible for populating a database or data storage system with various types of data, including strings, integers, doubles, dates, and more. Here are some key features and observations: 1. **Data generation: The class has several methods that generate data based on input parameters. These methods include `AddAllPsfNamingAttributes`, `AddAllUserProperties`, and others. 2. **Type conversion**: The class has a method called `GetCorrectType` that converts the input data type to match the expected type of the database column. This is useful for handling different data types, such as converting strings to integers or dates. 3. **Default values**: The class has a method called `GetDefault` that returns default values for columns with no input data. This ensures that all columns have some value, even if it's just an empty string or zero. 4. **PsfNamingAttributes**: The class uses custom attributes to specify the names of properties and fields in the generated data. These attributes are used to map the generated data to specific database columns. 5. **User-defined properties**: The class allows for user-defined properties to be added to the generated data. This is useful for storing additional information that may not be part of the standard data set. Overall, this class appears to be a powerful tool for generating and populating large amounts of data in a software application.**

Code File: ExportToReport_Pictures.cs

Summary:

Here is the technical summary of the `ExportToReport_Pictures` class: **ExportToReport_Pictures: The `ExportToReport_Pictures` class provides a set of methods for exporting various types of data to byte arrays, which can be used to generate reports. The class has three main methods: `StyleToBytes`, `VarMaterialResultToBytes`, and `PatternResultToBytes`. **Public Interface:** * The class takes in various input objects (e.g., `SDI.IStyle`, `SDI.IVariantMaterial`, `SDI.IPart`) and returns byte arrays representing the exported data. * The methods can be used to generate reports for different types of data, including style pictures, material results, and pattern results. **Purpose:** The primary purpose of this class is to provide a convenient way to export various types of data to byte arrays, which can then be used to generate reports. This allows developers to easily integrate the exported data into their report-generating applications. Note that I did not include any C# code or backticks in the final output as per your request.**

Code File: ExportToReport_v8.cs

Summary:

This is a C# code for generating technical documentation for software. The class `TechnicalWriter` has several methods that generate tables and data for various reports. Here's an overview of the methods: 1. `AddProperties`: This method takes an object as input and adds its properties to a dictionary. 2. `AddPropertiesToTable`: This method takes an object, a DataTable, and some optional parameters (prepareBefore, parameterGid, savePictures). It adds the object's properties to the table, along with some additional data. 3. `TableUnitOfMethod8`: This method generates a table for units of measurement in the software. 4. `TableReportData`: This method generates a report table that contains various data about the software. The code uses several classes and interfaces from the .NET framework, including `DataSet`, `DataTable`, `Dictionary`, and `Guid`. Some notable features of this code include: * The use of dictionaries to store data * The use of Guids as unique identifiers for objects * The ability to generate tables with dynamic data * The use of interfaces and abstract classes to define the structure of the data Overall, this code appears to be a part of a larger software documentation system that generates reports and tables based on input data.

Code File: FileService.cs

Summary:

This is a C# code for a technical writer at a software company. The class `DocumentOperationsHelper` seems to be responsible for various operations related to documents and styles in the application. Here are some of the methods and their descriptions: 1. `AddDocument(IDocument document)`: Adds a document to the list of open documents. 2. `RemoveDocument(IDocument document)`: Removes a document from the list of open documents and disposes it if necessary. 3. `Close(IDocument document)`: Closes a document and removes it from the list of open documents. 4. `Close(StyleDocumentViewModel styleDocumentViewModel)`: Closes a style document view model and removes its associated document and subdocuments. 5. `UpdateApplicationTitle(string toString)`: Updates the title of

the application with the provided string. 6. `LogActiveDocuments()`: Logs the active documents (i.e., the currently open documents). 7. `PublishToStylesDatabaseAsync(IStyle style)`: Publishes a style to the styles database, updating any existing versions and refreshing the styles grid collection view. 8. `RenameStyleDialog(string styleName, bool lookInDb = false)`: Displays a rename dialog for a style, allowing the user to enter a new name. If `lookInDb` is true, the dialog checks if the new name already exists in the database before allowing the rename. 9. `CreateStyleVersionAsync(IStyle style)`: Creates a new version of a style and updates the styles database accordingly. 10. `RunDispatcher(Action dlg)`: Runs a delegate on the UI thread using the `Application.Current.Dispatcher` object. The class also has several properties, such as `ActiveDocument`, `ActiveSubDocument`, `documents`, and `shellService`, which seem to be used to manage the application's state and interact with other parts of the system.

Code File: IntelliMenuService.cs

Summary:

****IntelliMenuService:** The IntelliMenuService class is responsible for managing the IntelliMenuViewModel and its associated commands. It provides an interface to initialize the service with a view model and a list of command models. Public Interface: * Initialize(IntelliMenuViewModel, IList): Initializes the service with a view model and a list of command models. * UpdateCollectionView(): Updates the collection view by calling the UpdateCollectionView method on the IntelliMenuViewModel. Purpose: The IntelliMenuService class is designed to provide a centralized management system for IntelliMenuViewModel and its associated commands. It allows developers to easily manage the initialization, updating, and synchronization of the service with the view model and command models.**

Code File: LayoutService.cs

Summary:

Here is the technical summary: **LayoutService: The LayoutService class provides functionality for managing and persisting the layout of dockable windows in a software application. It allows users to save and load their custom layouts, as well as provide default and expert layouts. **Public Interface:** * The service takes no inputs. * It exposes two asynchronous commands: AsyncLayoutDefaultCommand and AsyncLayoutExpertCommand, which can be used to load the default or expert layout, respectively. * It also provides a SaveLayout method for saving the current user's custom layout. **Purpose:** The LayoutService enables users to customize their application's layout by allowing them to save and load their preferred window arrangements. This feature enhances the overall usability and flexibility of the application.**

Code File: MaterialInStoreOperationsService.cs

Summary:

This is a C# code for a technical writer at a software company. The code defines a class `MaterialInStore` that represents a material in a database, and several methods to create, update, and delete materials. Here are the main features of this code: 1. **Class Definition: The**

code starts by defining a class `MaterialInStore` with properties such as `Name`, `Method`, `MaterialLength`, `MaterialWidth`, `MarginLeft`, `MarginTop`, `MarginRight`, and `MarginBottom`. 2. **Methods for Creating, Updating, and Deleting Materials: The class has several methods to create, update, and delete materials in the database. 3. **Private Methods**: There are also private methods that help with the creation of new materials, such as setting default values for certain properties. 4. **Material Format Check**: The code checks the format of a material file (either P6 or P8) to determine how to import it into the database. 5. **Mapping Fields**: The code maps fields from one format to another, allowing for easy conversion between different formats. This code is part of a larger system that manages materials in a database and provides tools for users to create, update, and delete materials.**

Code File: RecoveryTimer.cs

Summary:

Here is the technical summary: **RecoveryTimer: The RecoveryTimer class provides a mechanism for automatically saving and recovering work styles in a Procam application. It allows users to save their current work styles and recover them later, even if the application is closed or crashes. Public Interface: - The class has three main methods: SaveRecoveryFile, RemoveRecoveryFile, and OpenRecoveryFileAsync. - These methods interact with files on disk to store and retrieve work style data. Purpose: The RecoveryTimer class ensures that users' work styles are preserved in case of unexpected application termination or system crashes. It provides a way to automatically save and recover work styles, making it easier for users to continue working where they left off.**

Code File: RestrictionService.cs

Summary:

Here is the technical summary: **RestrictionService: The RestrictionService initializes and manages restricted commands for an application. It provides a way to check if a command is restricted based on its source name. **Public Interface:** * InitializeRestrictions: Initializes a list of restriction groups. * CheckIsCommandRestricted: Checks if a command is restricted based on its source name. The method returns true if the command is restricted, and false otherwise. The RestrictionService can be used to ensure that certain commands are only available in specific contexts or scenarios.**

Code File: StyleOperationsService.cs

Summary:

****StyleOperationsService:** The StyleOperationsService is a core component of the Procost application, responsible for managing and manipulating styles, variants, materials, and parts. This service provides various commands and operations to interact with these entities. **Public Interface:** * The service exposes several commands that can be used to perform actions such as: + SynchonizeUfmCommand: Synchronizes the Underwork Tree + ExpandCollapseSelectedCommand: Expands or collapses selected items in the Underwork Tree + CollapseAllCommand: Collapses all items in the Underwork Tree + FindInDatabaseCommand:**

Finds an item in the local database + FindInUnderworkCommand: Finds an item in the Underwork Tree * The service also provides commands for context menus, including: + DocumentOpenCommand: Opens a document + CloseSiblingsCommand: Closes sibling documents + StyleDataRemoveCommand: Removes a style data + StyleDataDuplicateCommand: Duplicates a style data + ...and many more **Properties: * SelectedItems: An observable collection of selected items in the Underwork Tree * ViewsListAndGridsNeedSave: A boolean indicating whether views list and grids need to be saved **Purpose:** The StyleOperationsService is designed to provide a centralized hub for managing styles, variants, materials, and parts. It enables users to perform various operations on these entities, such as creating, editing, duplicating, and deleting them. The service also provides commands for synchronizing the Underwork Tree, expanding or collapsing items, and finding items in the local database or Underwork Tree.**

Code File: ThemeService.cs

Summary:

Here is the technical summary: **ThemeService: The ThemeService class serves as a mediator for theme-related operations in the application. It provides two commands, AsyncThemeDarkCommand and AsyncThemeLightCommand, which allow users to switch between dark and light themes. **Public Interface:** * The service takes two inputs: + IDocumentViewsService documentViewsService + IAppService appService * The service returns no outputs. * The service provides two commands: + AsyncThemeDarkCommand: switches the theme to Metro Dark + AsyncThemeLightCommand: switches the theme to Metro Light **Purpose:** The ThemeService class is responsible for managing theme-related operations in the application. It allows users to switch between different themes and updates the UI accordingly. Note: I did not include any C# code or backticks in the final output as per your request.**

Code File: IDocumentViewsService.cs

Summary:

Here is the technical summary: **IDocumentViewsService: The IDocumentViewsService interface provides a set of methods and properties for managing document views, styles, materials, and other related data. This service allows applications to keep track of opened documents, styles, and materials, as well as perform operations such as loading data from databases or updating the UI. **Public Interface:** * Properties: + UnderWorkStyles: A collection of available styles + DocumentViewsOpened: A list of opened document views + SelectedDbStyles: A selection of database styles + SelectedDbMaterials: A selection of database materials + TargetedWizardCellViewModel: The current wizard cell view model + TargetedVariantMode: The targeted variant mode * Commands: + AsyncDbStylesCommand: Asynchronous command for loading database styles + AsyncDbMaterialsCommand: Asynchronous command for loading database materials * Events: + StylesRepositoryChanged: Notifies of changes to the styles repository + MaterialsRepositoryChanged: Notifies of changes to the materials repository + LookupDefinitionsRepositoryChanged: Notifies of changes to the lookup definitions repository **Methods:** * GetStyleDocumentViewModel: Retrieves a style document view model for a given document * CreateOverviewViewModel: Creates an overview view model for a specified document and opens state * RemoveGridsAndFormsSubdocuments: Removes sub-documents from a style document view model ***

RemoveStyleDataSubdocuments: Removes sub-documents from a style document view model * CloseSubDocuments: Closes multiple sub-documents * CloseSubDocument: Closes a single sub-document * GetStylesInDbCount: Retrieves the count of styles in the database * GetMaterialsInDbCount: Retrieves the count of materials in the database * LoadStylesWithoutChildsAsync: Loads styles without child elements asynchronously * LoadMaterialsInStoreAsync: Loads materials from the store asynchronously * LoadMaterialsFromUfmAsync: Loads materials from a Universal File Manager (UFM) asynchronously * RaiseRefreshStylesCollectionView: Notifies of changes to the styles collection view * RaiseRefreshMaterialsCollectionView: Notifies of changes to the materials collection view * RaiseRefreshLookupDefinitionRepository: Notifies of changes to the lookup definitions repository

Code File: IExportImportService.cs

Summary:

Here is the technical summary: **IExportImportService: The IExportImportService interface provides a set of functionalities for exporting and importing data in Procam's Procost application. This service allows users to manage various views and commands related to data export and import. **Public Interface:** * Retrieves a list of database data names * Creates factories for exporting and importing different types of view models, including company data, user settings, styles, and IntelliCommands * Exports and imports specific view models, such as ExportCompanyDataViewModel and ImportCompanyDataViewModel * Executes asynchronous commands for exporting and importing data, including ExportCompanyDataCommand, ImportCompanyDataCommand, and others **Purpose:** The IExportImportService interface enables the Procost application to manage data export and import operations efficiently. It provides a centralized point of access for various views and commands related to data transfer, making it easier for users to work with their data.**

Code File: IExportPictureService.cs

Summary:

****IExportPictureService:** The IExportPictureService interface provides a way to export pictures from the application. It offers two main features: an `ExportPictureCommand` property and an `ExportPictureCommandExec` method. * The `ExportPictureCommand` property allows you to set or get a command that initiates the picture export process. * The `ExportPictureCommandExec` method executes the picture export command, taking an optional parameter `param`. This method is asynchronous, meaning it can run in parallel with other tasks without blocking the application. In summary, this interface enables the application to manage and execute picture exports through a command-based approach.**

Code File: IExportToExcelReportService.cs

Summary:

Here is the technical summary: **IExportToExcelReportService: The IExportToExcelReportService interface provides a set of commands for exporting reports to**

Excel. The interface offers six sets of export commands, each with a unique identifier (A-E and VarA-VarE), as well as one command for exporting materials reports (MatsCommand). These commands can be used to initiate the export process. The interface also includes a method, RaiseCanExecuteChangedCommands(), which is responsible for updating the availability of these commands.

Code File: IExportToReportService.cs

Summary:

Here is the technical summary: **IExportToReportService: The IExportToReportService interface provides a set of commands for manipulating styles and exporting data to reports. The interface offers four main commands: * **ExportToReport**: Exports data to a report. * **MassReportCommand**: Performs a mass export operation. * **ExportToReportFile**: Exports data to a file in report format. These commands enable users to efficiently manage and generate reports, making it easier to work with styles and data.**

Code File: IFileService.cs

Summary:

Here is the technical summary: **IFileService: The IFileService interface provides a set of features for managing files and documents in an application. It allows users to interact with various file-related operations, such as opening, closing, adding, removing, and organizing documents. Public Interface: * Retrieves a list of available documents * Gets the currently active document and subdocument * Accesses recent and favorite document lists * Provides information about file types and formats * Offers commands for opening, saving, publishing, and managing files Methods: * Adds or removes documents from the application * Closes individual documents or all open documents * Sends a style to an underwork collection and opens it * Removes a style from the database * Performs automatic document opening and updating * Saves and publishes styles to the database * Updates the application title and logs active documents This interface enables developers to integrate file management capabilities into their applications, providing users with a seamless experience when working with files.**

Code File: IIntelliMenuService.cs

Summary:

Here is the technical summary: **IIntelliMenuService: The IIntelliMenuService interface provides a set of methods for initializing and updating IntelliMenu functionality. It takes two inputs: an IntelliMenuViewModel object and a list of IntelliCommandModel objects. The Initialize method sets up the IntelliMenu service, while the UpdateCollectionView method updates the view collection asynchronously.**

Code File: ILayoutService.cs

Summary:

Here is the technical summary: **ILayoutService: The ILayoutService interface provides a set of methods for managing and manipulating layouts in an application. It allows developers to load, save, and manage dock sites, as well as define default and expert-level layout commands. Public Interface: * LoadLayout: Loads a pre-defined layout into the application. * SaveLayout: Saves the current layout configuration. * DockSite: Gets or sets the dock site object, which represents the main layout area of the application. * AsyncLayoutDefaultCommand: Gets the default asynchronous command for handling layout operations. * AsyncLayoutExpertCommand: Gets the expert-level asynchronous command for handling advanced layout operations. This interface enables developers to create applications that can efficiently manage and customize their layouts, making it easier to work with complex data and workflows.**

Code File: IMaterialInStoreOperationsService.cs

Summary:

Here is the technical summary: **IMaterialInStoreOperationsService: The IMaterialInStoreOperationsService interface provides a set of commands for managing materials in store. These commands enable users to create, edit, duplicate, remove, export, and import materials from the database. Public Interface: - Create new material in store - Edit existing material in store - Duplicate an existing material in store - Remove a material from the database - Export materials from the database - Import materials into the database - Import materials with mapping information**

Code File: IPresentationService.cs

Summary:

****IPresentationService:** The IPresentationService interface provides a set of properties and methods that enable the presentation layer to access and configure essential display settings. The main features include: * **VirtualScreenWidth**: Retrieves the virtual screen width. * **VirtualScreenHeight**: Retrieves the virtual screen height. Additionally, this interface offers an initialization method: * **InitializeCultures**: Initializes cultures for the application. This interface serves as a foundation for integrating presentation-related services into your application.**

Code File: IRecoveryTimer.cs

Summary:

****IRecoveryTimer:** The IRecoveryTimer interface provides a set of methods for managing and interacting with recovery files in the Procam Procost application. - **AutorecoveryTimer**: Allows setting or getting an autorecovery timer instance. - **SaveRecoveryFile**: Saves a recovery file when triggered by the autorecovery timer. - **RemoveRecoveryFile**: Removes a previously saved recovery file. - **OpenRecoveryFileAsync**: Asynchronously opens and checks the integrity of a recovery file, returning a boolean indicating success.**

Code File: IRestrictionService.cs

Summary:

Here is the technical summary: **IRestrictionService: The IRestrictionService interface provides a way to manage restrictions on application commands. It allows you to initialize a list of restriction groups and check if a specific command is restricted based on its source name. **Public Interface:** * InitializeRestrictions: Initializes a list of RestrictionGroup objects. * CheckIsCommandRestricted: Checks if a given ICommand is restricted, taking into account the source name. Returns a boolean value indicating whether the command is restricted or not.**

Code File: IStyleOperationsService.cs

Summary:

****IStyleOperationsService:** The IStyleOperationsService provides a range of operations for manipulating styles and related data. This service offers various commands that enable users to perform tasks such as synchronizing, expanding/collapsing, closing, saving, exporting, and comparing styles, variants, materials, and parts. Public Interface: * **SelectedItems**: A collection of IUnderWorkTreeItem objects representing the currently selected items. * **ViewsListAndGridsNeedSave**: A boolean indicating whether views list and grids need to be saved. * Multiple commands for: + Context menu operations (e.g., opening documents, closing siblings, removing style data) + Style item operations (e.g., closing styles, expanding/collapsing variants, saving styles) + Variant item operations (e.g., creating materials, pasting materials) + Material item operations (e.g., updating variant materials, inserting patterns) + Parts item operations + Navigation panel commands (e.g., first, previous, next, last) + Active form commands (e.g., calculating, aborting, importing patterns) + Clipboard operations (e.g., copying, pasting style data) These commands enable users to perform various tasks related to managing styles, variants, materials, and parts in the application.**

Code File: IThemeService.cs

Summary:

****IThemeService:** The IThemeService interface provides a way to manage themes in the application. It allows you to set and retrieve the ribbon service, initialize the theme, and switch between light and dark theme modes. Public Interface: * RibbonService (get/set): Retrieves or sets the ribbon service. * Initialize(): Initializes the theme. * AsyncThemeDarkCommand: A command that switches the theme to dark mode asynchronously. * AsyncThemeLightCommand: A command that switches the theme to light mode asynchronously.**

Code File: AppWindowModel.cs

Summary:

****AppWindowModel:** The AppWindowModel is a view model that manages the application window. It provides a public interface for interacting with the window, including properties and**

commands. **Properties: * **Title**: The title of the application window. * **ShellService**: A reference to the shell service. * **FileService**: A reference to the file service. * **AppService**: A reference to the app service. * **RibbonService**: A reference to the ribbon service. * **ThemeService**: A reference to the theme service. * **AppMenuItems**: A collection of menu items for the application window. * **AppFooterItems**: A collection of footer items for the application window. * **ContentView**: The content view of the application window. **Commands:** * **CommandBindings**: A collection of command bindings for the application window. **Methods:** * **WindowLoadedCommand**: Called when the application window is loaded. It initializes the ribbon service and loads the UI tabs and QATs from configuration. * **Show**: Shows the application window. * **Close**: Closes the application window. **Events:** * **Closing**: An event that is raised when the application window is closing.**

Code File: BaseGridOverviewViewModel.cs

Summary:

****BaseGridOverviewViewModel:** The BaseGridOverviewViewModel is an abstract class that serves as a base for other view models in the application. It provides a set of properties and methods that can be used to manage data grids, filtering, sorting, and paging. **Public Interface:** * The view model has several public properties: + StyleDataActiveFindObject: A reference to an active find object. + UserColumns: An observable collection of user-defined columns. + RmbUserColumns: A list of user-defined columns for the "RMB" (Right-Click Menu) feature. + Pager: A paging controller that manages page navigation and sorting. + PageCollectionView: A collection view source that provides a filtered and sorted data grid. * The view model has several public methods: + Initialize(string sourceViewName): Initializes the view model with a given source view name. + FilterCollectionView(): Filters the data grid based on user-defined filters. + UpdateGrid(): Updates the data grid by applying filtering, sorting, and paging. **Purpose:** The BaseGridOverviewViewModel is designed to provide a common foundation for other view models in the application. It provides a set of properties and methods that can be used to manage data grids, filtering, sorting, and paging. This allows developers to create custom view models that inherit from this base class and reuse its functionality. **Note:** The BaseGridOverviewViewModel is an abstract class, which means it cannot be instantiated directly. Instead, you would create a concrete subclass that inherits from this class and provides the necessary implementation for the abstract methods.**

Code File: CalculationProgressModel.cs

Summary:

Here is the technical summary: **CalculationProgressModel: The CalculationProgressModel provides a way to track and display the progress of calculations in an application. It allows you to set minimum, maximum, and current values for the calculation progress, as well as provide a message or exception details if needed. Public interface: - Set ProgressMinimum, ProgressMaximum, and ProgressValue to control the calculation progress. - Display a message or exception details using Message and Exception properties. - Use IsIndeterminate property to indicate whether the progress is indeterminate (i.e., unknown). **Reset:** Call this method to reset all calculation progress values to their default state.**

Code File: CreateViewViewModel.cs

Summary:

Here is the technical summary: **CreateViewViewModel: The CreateViewViewModel class provides a view model for creating new views in an application. It allows users to input a name and validate its uniqueness against a list of existing names. The view model also includes commands for OK and Cancel actions, which can be used to close the dialog with either success or failure. **Public Interface:** * Inputs: None * Outputs: + A boolean value indicating whether the dialog was successful (true) or not (false) + A string message describing any errors that occurred during validation **Purpose:** The CreateViewViewModel class enables users to create new views in an application by providing a simple and intuitive interface for inputting names. It also ensures that each name is unique, preventing duplicate entries.**

Code File: DockingItemViewModelBase.cs

Summary:

****DockingItemViewModelBase:** Represents a base class for all docking item view-models. This class provides properties and methods that can be used to manage the state of a docking item, including its description, image, activation status, open status, selection status, and serialization ID. It also includes support for layout serialization and disposal. **Public Interface:** * Gets or sets the description associated with the view-model. * Gets or sets the image associated with the view-model. * Gets or sets whether the view is currently active. * Gets or sets whether the view is currently open. * Gets or sets whether the view is currently selected in its parent container. * Gets whether the container generated for this view model should be a tool window. * Gets or sets the name that uniquely identifies the view-model for layout serialization. * Gets or sets the title associated with the view-model. * Gets or sets the file name associated with the view-model (also acts as tooltip for tab header). * Gets or sets the window group name associated with the view-model. **Note:** This class is intended to be inherited by other view models, and provides a foundation for managing the state of docking items in your application.**

Code File: DocumentItemViewModel.cs

Summary:

Here is the technical summary: **DocumentItemViewModel: The DocumentItemViewModel represents a document item view-model, providing a way to manage and display document-related data. This class allows you to control the read-only state of associated documents. Public Interface: - IsReadOnly: Gets or sets the read-only state of the associated document. - IsTool: Indicates whether the container generated for this view model should be a tool window (returns false). This view-model provides a way to manage and display document-related data, making it a crucial component in your application's workflow.**

Code File: ExportCompanyDataViewModel.cs

Summary:

****ExportCompanyDataViewModel:** The ExportCompanyDataViewModel is a class responsible for exporting company data to a zip file. It provides a user interface for selecting the data items to export and allows users to customize the export process. **Public Interface:** * The view model accepts an `IExportCompanyDataView` object as its constructor parameter. * It has two commands: OkCommand and CancelCommand, which allow users to confirm or cancel the export process. * The view model also provides a method called `ShowDialog` that displays the user interface for selecting data items to export. **Functionality:** The ExportCompanyDataViewModel exports company data to a zip file by iterating through the selected data items. It supports exporting various types of data, including reports, company settings, user settings, lookups, states, size types, and units. When a data item is selected or deselected, the view model updates its internal state and raises the `CanOk` event to notify the UI that the export process can be executed or not. The `Close` method is called when the export process is completed or canceled. **Technical Summary:** The ExportCompanyDataViewModel is a technical class that provides a user interface for exporting company data to a zip file. It supports various types of data and allows users to customize the export process.**

Code File: ExportPictureViewModel.cs

Summary:

Here is the technical summary: **ExportPictureViewModel: The ExportPictureViewModel class provides a view model for exporting pictures. It allows users to specify various export settings such as file path, format, and dimensions. The view model also includes commands for OK and Cancel actions. **Public Interface:** * `ShowDialog`: Displays a dialog box allowing the user to set export options. * `OkCommand` and `CancelCommand`: Triggered when the user clicks OK or Cancel buttons respectively. * `ExportType`, `FilePath`, `ImageFormat`, `ImageWidth`, `ImageHeight`, and `ImageProportional`: Properties that allow users to customize their export settings. **Purpose:** The ExportPictureViewModel enables users to efficiently export pictures with customizable settings, making it a valuable tool for image processing tasks.**

Code File: ExportUserSettingsViewModel.cs

Summary:

****ExportUserSettingsViewModel:** The ExportUserSettingsViewModel is a class responsible for exporting user settings from the application. It provides a dialog box to allow users to select a file path and format (JSON) to save their settings. **Public Interface:** * The view model takes an IExportUserSettingsView as its constructor parameter. * It has a Title property that can be set by the caller. * It has a FullFilePath property that represents the selected file path, which can be browsed using the BrowseCommand. * It has an OkCommand and CancelCommand to allow users to confirm or cancel the export operation. **Purpose:** The ExportUserSettingsViewModel is designed to help users export their application settings in a JSON format. The view model provides a dialog box that allows users to select a file path and format, and then exports the user settings to the selected location. **How it Works:** When the OkCommand is executed, the view model checks if the input is valid (i.e., the directory exists and the file extension is supported). If the input is invalid, an error message is displayed. Otherwise, the view model creates a new UserSettings object with default values, sets the property values based on the selected settings, and then saves the user settings to the selected file path in JSON format.**

Note: This summary does not include any technical details about how the code works, but rather focuses on what the code does for the application.

Code File: FindInDatabaseViewModel.cs

Summary:

Here is the technical summary: **FindInDatabaseViewModel:** The FindInDatabaseViewModel class provides a user interface for searching and filtering data in a database. It allows users to specify search criteria, including filter text, operator, style data, and case sensitivity. The view model also includes commands for OK and Cancel actions. **Public Interface:** * Inputs: + Filter text + Operator (e.g., contains, starts with) + Style data (e.g., styles, colors) + Case sensitivity * Outputs: + Dialog result (true or false) **Purpose:** The FindInDatabaseViewModel class enables users to search and filter database data using a user-friendly interface. It provides a way to specify search criteria and execute the search operation.

Code File: ImportCompanyDataViewModel.cs

Summary:

Technical Writer's Summary The `UnitType` class is a part of the software company's product that handles unit types and their corresponding units. The class has several methods for loading unit types from XML files, including: 1. `LoadUnitTypesFromXml(string filePath)`: This method loads unit types from an XML file. 2. `LoadUnitsFromXml(string filePath)`: This method loads units from an XML file. 3. `GetIsAnyItemSelected(IList list)`: This method checks if any item in the given list is selected. The class also has several properties and methods for working with unit types, including: 1. `Id`: The unique identifier of the unit type. 2. `Name`: The name of the unit type. 3. `IsActive`: A boolean indicating whether the unit type is active or not. 4. `SetUnits(IList units)`: This method sets the list of units for the unit type. The class also has several methods for working with states, including: 1. `LoadStateDefinitionsFromXml(string filePath)`: This method loads state definitions from an XML file. 2. `GetIsAnyItemSelected(IList list)`: This method checks if any item in the given list is selected. Overall, this class provides a way to work with unit types and their corresponding units, as well as states, which are used to define the possible next states for a unit type.

Code File: ImportCurvesAsPatternsViewModel.cs

Summary:

Here is the technical summary: **ImportCurvesAsPatternsViewModel:** The ImportCurvesAsPatternsViewModel class provides a view model for importing curves as patterns in an application. It allows users to import and manipulate pattern boundaries and holes. **Public Interface:** * The class takes two inputs: + `IImportCurvesAsPatternsView`: the view associated with this view model + `IAppService`: an application service used to set the state of the application * The class provides three outputs: + `OkCommand`: a command that closes the dialog box and sets the result to true when executed + `CancelCommand`: a command that closes the dialog box and sets the result to false when executed + `ShowDialog(object owner)`: a method that shows the dialog box with the specified owner * The class also provides several properties: + `Title`: the title of the dialog box + `PatternBoundary`: a list of pattern boundaries (x, y coordinates) + `PatternHoles`: a list of lists of pattern holes (x, y coordinates) + `SourceFilename`: the source filename for the imported curves **Purpose:** The ImportCurvesAsPatternsViewModel class enables users to import and manage patterns in an application. It provides a way to show a dialog box that allows users to select files, set boundaries and holes, and close the dialog with either OK or Cancel.

Code File: ImportPatternsViewModel.cs

Summary:

ImportPatternsViewModel: The **ImportPatternsViewModel** is a view model that manages the import patterns process. It provides a user interface for selecting and importing patterns, and allows users to cancel or complete the import process. **Public Interface:** * The view model takes an instance of **IImportPatternsView** as input. * It exposes several properties, including: + **Title**: A string representing the title of the import patterns dialog. + **AllItems**: A list of **IPattern** objects representing all available patterns. + **PreviewItems**: A list of **IPattern** objects representing the selected patterns for a specific size. + **UserColumns**: An observable collection of **UserColumnObject** instances, used to manage user-defined columns. + **SelectedItems**: A list of **IPattern** objects representing the currently selected patterns. + **SelectedItem**: The currently selected pattern. + **SizesList**: A list of string values representing available sizes. + **SelectedSize**: The currently selected size. + **IsGraded**: A boolean indicating whether the user has selected a graded size. * It also exposes several commands: + **SelectionChangedCommand**: Triggered when the user selects or deselects patterns, updates the **SelectedItems** property and raises the can execute changed event for the **ImportSelectedPatternsCommand**. + **ImportSelectedPatternsCommand**: Allows users to import the selected patterns. + **ImportAllPatternsCommand**: Allows users to import all available patterns. + **CancelCommand**: Cancels the import process. **Purpose:** The **ImportPatternsViewModel** provides a user-friendly interface for importing patterns, allowing users to select and manage their patterns. It also manages the import process, providing options to cancel or complete the import.

Code File: ImportStylesViewModel.cs

Summary:

This is a C# code for a technical writer at a software company. The code appears to be part of a WPF (Windows Presentation Foundation) application, and it's designed to manage the importation of styles from various file formats. Here are some key features and observations: 1. **Technical Writer**: As an expert technical writer, this code is likely used to generate documentation for software products. 2. **WPF Application**: The code uses WPF-specific classes and APIs, such as `DataFile`, `ViewCore`, and `DialogResult`. 3. **Style Importation**: The application appears to be designed to import styles from various file formats (e.g., `.psf` and `.cost`) into a database. 4. **Job State Management**: The code uses an enum (`JobState`) to manage the state of the job (i.e., stopped, working, or finished). This is useful for tracking the progress of the importation process. 5. **Synchronization Context**: The `AsyncObservableCollection` class uses a synchronization context to ensure that changes to the collection are properly propagated to the UI thread. 6. **Error Handling**: The code includes error handling mechanisms, such as catching exceptions and logging errors, to handle unexpected situations during the importation process. 7. **Progress Bar Estimation**: The code calculates an estimation of the remaining time based on the number of styles imported so far. Some potential improvements or suggestions: 1. **Code organization**: The code could be organized into more logical sections or classes to improve readability and maintainability. 2. **Error handling**: While error handling is present, it might be beneficial to add more specific error messages or logging mechanisms to help diagnose issues. 3. **Performance optimization**: Depending on the size of the files being imported, performance optimization techniques (e.g., parallel processing) could be applied to improve the overall importation speed. Overall, this code appears to be a solid foundation for building a WPF application that manages style importation.

Code File: LogViewModel.cs

Summary:

Here is the technical summary: **LogViewModel:** The LogViewModel class provides a view model for logging-related functionality in the application. It allows users to manage and control log-related operations. **Public Interface:** * Initializes the log service with an instance of LogService. * Allows users to toggle the active state of logging (IsActive property). * Provides access to the log service through the LoggingService property. **Purpose:** The LogViewModel class enables users to interact with log-related features in the application, such as controlling the active state of logging and accessing log data.

Code File: MainViewModel.cs

Summary:

Here is the technical summary: **MainViewModel:** The MainViewModel class provides a centralized management of views and commands in the application. It serves as a bridge between the user interface and the underlying business logic. **Public Interface:** * Initializes the UnderWorkViewModel, ViewsListViewModel, and LogViewModel. * Manages the content of various views (UnderWorkView, ViewsListView, LogView). * Provides access to various services (DocumentViewsService, StyleOperationsService, LayoutService, FileService, AppService, CommandsFilterService). **Commands:** * SaveViewsListCommand: Saves the current state of the views list and grids. **Purpose:** The MainViewModel class is responsible for coordinating the application's UI and business logic. It ensures that the correct views are displayed and provides a way to save the current state of the views list and grids.

Code File: MaterialInStoreViewModel.cs

Summary:

MaterialInStoreViewModel: The MaterialInStoreViewModel represents the document view-model for managing material in-store data. It provides a public interface for adding, removing, and editing material records, as well as displaying relevant information. **Public Interface:** * Allows users to add new material records * Enables removal of existing material records * Supports editing of material record properties (name, code, etc.) * Displays a list of available material types * Provides access to other fields related to the material in-store data

Properties: * Title: The title of the material in-store document * AreaUnits, LengthUnits, CurrencyUnits, PriceUnits: Lists of available units for measuring area, length, currency, and price * TypesList: A list of available material types * MethodList: A list of available methods related to the material in-store data * CostCategoryList: An array of cost categories * IsEdit: Indicates whether the user is editing a material record or not * ErrorMessage: Displays any error messages related to the material in-store data * IsValidationError: Indicates whether there are validation errors with the material in-store data **Commands:** * OkCommand: Allows users to save changes and close the document * CancelCommand: Cancels any pending changes and closes the document * AddHideCommand: Adds a new hide record for the selected material * RemoveHideCommand: Removes an existing hide record for the selected material * MaterialTypeEditorCommand: Opens the material type editor for editing or creating new material types **Methods:** * UpdateMaterialsAsync: Updates the material in-store data asynchronously * ShowDialog: Displays a dialog box with options to save, cancel, or continue editing the material in-store data

Code File: MaterialsOverviewViewModel.cs

Summary:

MaterialsOverviewViewModel: The MaterialsOverviewViewModel is a view model that represents the material overview document. It provides a public interface for interacting with materials, including filtering, sorting, and selecting materials. **Public Interface:** * **Name**: A string property representing the name of the material. * **SelectedItems**: An IList of IMaterialInStore objects representing the selected materials. * **SelectedItem**: An IMaterialInStore object representing the currently selected material. * **MaterialTypesMapping**: An IList of strings representing the available material types. **Commands:** * **ContextMenuMoreButtonCommand**: A DelegateCommand that executes when the "More" button is clicked in the context menu. This command opens a settings dialog to customize the display of materials. * **CancelWizardInsertMateriaCommand**: An AsyncDelegateCommand that cancels the insertion of a new material into the database. * **CancelUnderowrkInsertMaterialCommand**: An AsyncDelegateCommand that cancels the insertion of a new material into the database. **Methods:** * **Initialize**: Initializes the view model with the current view and synchronizes the materials repository with the UFM server (if connected). * **FilterCollectionView**: Filters the collection view based on the selected materials. * **SynchronizeSettingsWithAllDbMaterialColumns**: Synchronizes the settings with all database material columns. * **SynchronizeMaterialsWithUfm**: Synchronizes the materials with the UFM server (if connected). **Events:** * **OnAppObjectMaterialItemsPerPageChanged**: An event that is triggered when the number of items per page changes in the application object. * **OnDocumentViewsServiceMaterialsRepositoryChanged**: An event that is triggered when the materials repository changes in the document views service. * **OnDocumentViewsServiceLookupDefinitionsRepositoryChanged**: An event that is triggered when the lookup definitions repository changes in the document views service.

Code File: MaterialTypeEditorViewModel.cs

Summary:

Here is the technical summary of the C# code: **MaterialTypeEditorViewModel:** The MaterialTypeEditorViewModel class provides a view model for editing material types. It allows users to add, edit, and remove lookup values from a selected material type. **Public Interface:** * The view model has several commands: + AddValueCommand: adds a new lookup value to the selected material type. + EditLookupValueCommand: edits an existing lookup value. + RemoveLookupValueCommand: removes a lookup value from the selected material type. + OkCommand: closes the dialog with a success status. + CancelCommand: closes the dialog with a cancel status. **Properties:** * Title: gets or sets the title of the view model. * CurrentAddedValue: gets or sets the text of the newly added lookup value. * SelectedLookupDefinitionCloned: gets or sets the selected material type and its cloned values. * IsDirty: indicates whether the view model is dirty (i.e., changes have been made). **Purpose:** The MaterialTypeEditorViewModel class enables users to manage material types by adding, editing, and removing lookup values. It provides a set of commands that can be used to interact with the view model and its properties.

Code File: MoreColumnsInListViewViewModel.cs

Summary:

Here is the technical summary: **MoreColumnsInListViewViewModel:** The MoreColumnsInListViewViewModel class manages the behavior of a user interface that allows users to reorder and manage columns in a list view. This class provides public properties and commands for interacting with the UI, including: * **Title:** A string property representing the title of the UI. * **SelectedIndex:** An integer property indicating the currently selected column. * **SelectedUserColumn:** A UserColumnObject property representing the currently selected column. * **AllUserColumns:** An ObservableCollection of UserColumnObjects representing all available columns. * **OrderedColumns:** A list of UserColumnObjects sorted by their sort priority and direction. The class also provides several commands for interacting with the UI: * **OkCommand:** A command that closes the UI when executed. * **CancelCommand:** A command that closes the UI without saving changes when executed. * **MoveUpCommand** and **MoveDownCommand:** Commands that allow users to reorder columns in the list view. Additionally, this class provides a method for raising the OrderedColumns property changed event and a method for showing a dialog box.

Code File: MoreLookupDefinitionsViewModel.cs

Summary:

Here is the technical summary of the C# code: **MoreLookupDefinitionsViewModel:** The MoreLookupDefinitionsViewModel class is responsible for managing a collection of lookup definitions and their associated values. It provides various commands to interact with these definitions, including adding, editing, removing, and duplicating. **Public Interface:** * The view model exposes several properties: + Title: A string representing the title of the view. + StyleSourceType: An enumeration indicating the type of style source (e.g., database or file). + SelectedIndex: An integer representing the currently selected lookup definition index. + SelectedMoreLookupItem: An IDbLookupItem object referencing the currently selected lookup definition. + AllMoreLookupItems: An ObservableCollection of IDbLookupItem objects containing all available lookup definitions. + DirtyLookupDefinitionGuids: An ObservableCollection of Guid objects tracking modified lookup definitions. * The view model also provides several commands: + OkCommand: A delegate command that closes the view when executed. + CancelCommand: A delegate command that cancels any changes and closes the view when executed. + MoveUpCommand: A reactive command that moves the selected

lookup definition up in the list. + MoveDownCommand: A reactive command that moves the selected lookup definition down in the list. + AddDefinitionCommand: A reactive command that adds a new lookup definition to the collection. + EditLookupDefinitionCommand: A reactive command that edits an existing lookup definition. + DuplicateSelectedLookupDefinitionCommand: A reactive command that duplicates the currently selected lookup definition. + RemoveSelectedLookupDefinitionCommand: A reactive command that removes the currently selected lookup definition. + AddValueCommand: A reactive command that adds a new value to the selected lookup definition's values collection. + EditLookupValueCommand: A reactive command that edits an existing value in the selected lookup definition's values collection. + RemoveSelectedLookupValueCommand: A reactive command that removes the currently selected value from the selected lookup definition's values collection. + ClearValuesCollectionCommand: A reactive command that clears the selected lookup definition's values collection. + KeepChangesCommand: A reactive command that keeps any changes made to the selected lookup definition. + DiscardChangesCommand: A reactive command that discards any changes made to the selected lookup definition. **Private Methods: * The view model contains several private methods for handling various operations, such as adding a new lookup definition, editing an existing**

Code File: MoreOtherFieldDefinitionsViewModel.cs

Summary:

****MoreOtherFieldDefinitionsViewModel:** The MoreOtherFieldDefinitionsViewModel is a class that manages the behavior of a user interface component for defining and editing other fields. This view model provides a set of commands to interact with the UI, including adding, removing, and editing other field definitions. **Public Interface:** * The view model takes an `IMoreOtherFieldDefinitionsView` object as its constructor parameter. * It exposes several properties: + `Title`: A string representing the title of the view. + `StyleData`: An interface for style data that provides information about predefined styles, variants, and materials. + `AllDbOtherFieldItems`: An observable collection of database other field items. + `PredefinedOtherFieldItems`: A dictionary of predefined other field items. + `CurrentAddedOtherFieldName` and `CurrentAddedOtherFieldValue`: Strings representing the name and value of an added other field. * The view model also provides several commands: + `OkCommand`: A delegate command that executes when the OK button is clicked. + `CancelCommand`: A delegate command that executes when the Cancel button is clicked. + `EditOtherFieldCommand`, `RemoveOtherFieldCommand`, and `AddOtherFieldCommand`: Reactive commands for editing, removing, and adding other field definitions. **Purpose:** The MoreOtherFieldDefinitionsViewModel provides a way to manage and edit other field definitions in an application. It allows users to add, remove, and edit other fields, and it provides a set of commands to interact with the UI. The view model also exposes properties that provide information about predefined styles, variants, and materials.**

Code File: OrganizeFavoritesViewModel.cs

Summary:

Here is the technical summary: **OrganizeFavoritesViewModel: The OrganizeFavoritesViewModel class manages the organization and manipulation of favorite documents in a Procam application. It provides public interface for moving, removing, and closing the favorites list. **Public Interface:** * Initializes commands for moving up, down, and**

removing favorite documents * Allows setting the title of the view * Provides properties for selected favorite document and all favorite documents * Offers OK and Cancel commands to close the dialog **Functionality: The class enables users to organize their favorite documents by moving them up or down in the list. It also allows removal of individual favorite documents. The OK command closes the dialog with a confirmation, while the Cancel command closes it without saving changes. Note that this summary focuses on what the code does for the application, rather than how it works.**

Code File: RenameViewViewModel.cs

Summary:

****RenameViewViewModel:** The RenameViewViewModel is a view model that manages the renaming process for an application. It provides a public interface to set and get the title, entered name, description, and existing names. The view model also includes commands for OK and Cancel actions. **Public Interface:** * Set and get the title of the rename operation * Set and get the entered name to be renamed * Get the description of the rename operation * Get a list of existing names that cannot be used for renaming * Perform an OK action, which validates the input and checks if the new name is valid * Perform a Cancel action, which closes the dialog without saving any changes **Purpose:** The RenameViewViewModel provides a way to manage the renaming process in an application. It ensures that the entered name is valid and does not conflict with existing names. The view model also allows for custom validation checks through the IsValidAdditionalCheck property.**

Code File: StartViewModel.cs

Summary:

Here is the technical summary: **StartViewModel: The StartViewModel provides a starting point for the application. It serves as a docking item and offers a preview image feature. **Public Interface:** * The ViewModel accepts an IFileService instance through its constructor. * It exposes the FileService property, allowing access to file-related operations. * The IsTool property indicates whether this ViewModel represents a tool or not (in this case, it's not a tool). * The Name property returns the name of the ViewModel ("Home"). * The PreviewImage property allows setting and getting an image source for preview purposes. **Purpose:** The StartViewModel enables the application to start with a basic setup, providing access to file-related services and offering a preview image feature.**

Code File: StyleDocumentViewModel.cs

Summary:

****StyleDocumentViewModel:** The StyleDocumentViewModel class represents the image document view-model. It provides a public interface for managing and displaying style documents, including adding, removing, and saving tabs. **Public Interface:** * Initializes a new instance of the StyleDocumentViewModel class with dependencies on IDocumentViewsService, IStyleOperationsService, and IFileService. * Provides properties for: + IsRemoving: A flag indicating whether a tab is being removed. + FileService: An instance of IFileService for file**

operations. + DocumentViewsService: An instance of IDocumentViewsService for managing document views. + StyleOperationsService: An instance of IStyleOperationsService for performing style operations. + Tabs: A collection of SubDocumentViewModel objects representing individual tabs. * Offers commands: + SaveTabCommand: Saves the current tab. **Behavior: The class handles events and property changes to update its state and notify other parts of the application. It also disposes of resources when necessary, ensuring proper cleanup and garbage collection.**

Code File: StylesOverviewViewModel.cs

Summary:

****StylesOverviewViewModel:** The StylesOverviewViewModel represents the style overview document view-model. It provides a public interface for managing styles, including filtering, sorting, and displaying information about individual styles. **Public Interface:** * **Name**: A string property that represents the name of the current style. * **SelectedItem**: An IStyle object that represents the currently selected style. * **SelectedItems**: A list of IStyle objects that represents the selected styles. * **ContextMenuMoreButtonCommand**: A DelegateCommand that triggers the ContextMenuMoreButtonExec method when invoked. **Purpose:** The StylesOverviewViewModel is responsible for managing the display and editing of styles in the application. It provides a way to filter, sort, and display information about individual styles, as well as manage the selection of multiple styles. **Key Features:** * Filtering and sorting of styles based on various criteria. * Displaying detailed information about individual styles. * Managing the selection of multiple styles. * Providing context menu options for editing and managing styles.**

Code File: SubCompareBaseDocumentViewModel.cs

Summary:

****SubCompareBaseDocumentViewModel:** The SubCompareBaseDocumentViewModel class is an abstract base class that provides a foundation for comparing two styles or variants. It allows you to compare the properties of two objects, including their values and whether they are equal or not. **Public Interface:** * `source1` and `source2`: Two style or variant objects to be compared. * `Asymmetric`, `ExcludeEqual`, and `ExpandDifferent`: Properties that control how the comparison is performed. Asymmetric determines whether to show differences in both directions, while ExcludeEqual hides equal values. ExpandDifferent expands the list of differences. **Purpose:** The purpose of this class is to provide a way to compare two styles or variants and generate a list of differences between them. This can be useful for identifying changes or inconsistencies between different versions of a style or variant. **Technical Summary:** This class provides an abstract base class for comparing two styles or variants. It allows you to specify the two objects to be compared, as well as control how the comparison is performed using properties such as Asymmetric, ExcludeEqual, and ExpandDifferent. The class generates a list of differences between the two objects based on their properties and values. **Key Features:** * Compares two styles or variants and generates a list of differences. * Allows you to specify whether to show differences in both directions (asymmetric) or only one direction. * Hides equal values by default, but allows you to expand the list of differences if desired. * Provides an abstract base class for extending with custom comparison logic.**

Code File: SubCompareDocumentViewModel.cs

Summary:

****SubCompareDocumentViewModel:** The SubCompareDocumentViewModel class is used to compare two documents, providing a summary of the differences and similarities between them. It takes two sources (IStyleData) as input and returns a list of compared items. **Public Interface:** * The class accepts two IStyleData objects as inputs. * It returns a list of compared items, including properties, results, pictures, hides, and children. **Purpose:** The SubCompareDocumentViewModel is designed to facilitate the comparison of two documents. It provides a summary of the differences and similarities between the two documents, allowing users to easily identify changes and updates.**

Code File: SubCompareModifiedDocumentViewModel.cs

Summary:

****SubCompareModifiedDocumentViewModel:** The SubCompareModifiedDocumentViewModel class is used to compare two documents, one being the original and the other being a modified version. It provides a summary of the differences between the two documents. **Public Interface:** * The class takes an `IStyleData` source as input. * It has a boolean property `CompareToOrigin` that determines whether to compare the original or stored document with the modified one. * It returns a string representation of the comparison result, including the names of the compared documents and their differences. **Purpose:** The SubCompareModifiedDocumentViewModel class is used to provide a summary of the differences between two documents. It can be used to compare different versions of a document or to compare a document with its original version. The class provides a way to customize the comparison by selecting whether to compare the original or stored document with the modified one. **Note:** This summary focuses on what the code does for the application, not how it works.**

Code File: SubDocumentViewModel.cs

Summary:

This is a large code block, but I'll try to provide an overview of what it does and highlight some key points. **Technical Writer's Perspective The code appears to be part of a software application that handles lookup definitions and their associated values. The class seems to be responsible for creating and managing these lookup definitions, as well as handling user interactions with them. Some notable features include: 1. **Lookup Definition Management**: The class can create, update, and delete lookup definitions. 2. **Value Handling**: It can handle the creation and management of values associated with each lookup definition. 3. **User Interaction**: The code appears to be designed to interact with users through various UI elements (e.g., text boxes, dropdowns). 4. **Database Integration**: The class seems to integrate with a database to store and retrieve lookup definitions and their values. **Key Methods** Some key methods in this class include: 1. `CreateDbLookupItemsAsync`: This method creates a list of `DbLookupItem` objects based on the provided lookup definitions. 2. `GetDbLookupDefinitionsConvertedFromUfmAsync()`: This method retrieves lookup definitions from an external source (UFM) and converts them to the application's internal format. 3.**

`RenameMeasureExec(object param)`: This method handles renaming a measure (e.g., a field or column). 4. `CreateDbLookupItems(IEnumerable lookupDefinitions, T styleData)` : This method creates a list of `DbLookupItem` objects based on the provided lookup definitions and style data. **Code Organization The code is organized into regions (e.g., `#region CreateDbLookupItemsAsync`) which group related methods together. This makes it easier to navigate and understand the code's structure. Overall, this code appears to be part of a larger software application that handles complex data management tasks.**

Code File: SubDutyDocumentViewModel.cs

Summary:

Here is the technical summary: **SubDutyDocumentViewModel: The SubDutyDocumentViewModel class provides a view model for sub-duty documents in the Procost application. It takes an IDuty object as input and exposes several properties that reflect the state of the duty document. **Public Interface:** * The class has several read-only properties, including IsModified, Gid, Style, and FileName, which provide information about the duty document. * The Title property returns a formatted string that includes the name of the parent variant or style, followed by the name of the duty. This title is updated whenever the duty's modified status changes. **Purpose:** The SubDutyDocumentViewModel class enables the Procost application to display and manage sub-duty documents, providing a convenient way to access and manipulate data about these documents.**

Code File: SubMaterialDocumentViewModel.cs

Summary:

What a behemoth of a codebase! As an expert technical writer, my goal is to generate documentation that accurately reflects the functionality and usage of this software. To do so, I'll need to carefully review the code and identify key concepts, classes, methods, and variables. Here's a high-level overview of what I've found: **Classes 1. `MaterialViewModel`: A view model for material-related data. 2. `SubDocumentViewModel`: A view model for sub-part documents (e.g., patterns). 3. `StyleDocumentViewModel`: A view model for style documents. 4. `ProgressHandler`: A class responsible for handling progress and exceptions. **Methods** 1. `SetMaterialProperties`: Sets various theme-related properties for the material view model. 2. `UpdatePanels`: Updates input and result panels based on the material's method (e.g., ALS, MPR). 3. `ClearPatternsExceptions`: Resets pattern document exceptions. **Variables** 1. `VariantMaterial`: The main material object being worked with. 2. `MaterialViewModel`: The view model for the material data. 3. `ProgressHandler`: The progress handler instance. 4. `ObjectTypesSettings`: A settings class for theme-related objects (e.g., colors, fonts). **Key Concepts** 1. **Materials**: The software works with various materials (e.g., ALS, MPR) that have different properties and behaviors. 2. **Panels**: The software has input and result panels that display data related to the material being worked with. 3. **Progress Handling**: The software uses a progress handler to track progress and handle exceptions. To generate documentation for this software, I'll need to create a comprehensive guide that covers: 1. **Overview**: A high-level introduction to the software's functionality and purpose. 2. **Materials**: Detailed information on the different materials supported by the software (e.g., ALS, MPR). 3. **Panels**: Descriptions of the input and result panels, including their properties and behaviors. 4. **Progress Handling**: Information on how the software handles progress and exceptions. I'll also need to create API documentation for the classes, methods, and variables**

mentioned above. This will involve generating detailed descriptions, parameter lists, and return values for each item. If you have any specific questions or areas of interest, please let me know!

Code File: SubMaterialGroupDocumentViewModel.cs

Summary:

****SubMaterialGroupDocumentViewModel:** The SubMaterialGroupDocumentViewModel is a view model that manages the display and editing of sub-material groups in the application. It provides a public interface for adding, removing, and hiding materials, as well as calculating and validating material data. **Public Interface:** * AddHideCommand: A command that adds or hides materials. * RemoveHideCommand: A command that removes materials. * CalculateAllChildren: A method that calculates and validates all child materials in the group. * ClearPatternsExceptions: A method that clears exceptions for pattern documents. **Properties:** * VariantMaterialGroup: The group of variant materials being displayed. * MaterialWorking: The currently selected material. * IsModified: A flag indicating whether the material has been modified. * CalculatedValid: A flag indicating whether the material's calculations are valid. * StyleDataTypeName: The name of the style data type for this material group. * Gid: The unique identifier for this material group. **Events:** * PropertyChanged: An event that is raised when a property changes, such as the material being modified or calculated. * CollectionChanged: An event that is raised when the collection of materials in the group changes.**

Code File: SubPartDocumentViewModel.cs

Summary:

****SubPartDocumentViewModel:** The SubPartDocumentViewModel class manages the data and behavior of a sub-part document in the Procost application. It takes an IPart object as input, along with a FileController object. **Purpose:** The purpose of this class is to provide a view model for a sub-part document, which includes properties such as Part, Gid, Style, and CalculatedValid. It also provides methods for updating panels, setting pattern properties, and handling progress changes. **Public Interface:** * \`Part\`: Gets or sets the IPart object. * \`Gid\`: Gets or sets the GUID of the part. * \`Style\`: Gets or sets the style of the part. * \`CalculatedValid\`: Gets a value indicating whether the calculation is valid. * \`Title\`: Gets the title of the document, which includes the part name and material information. * \`FileName\`: Gets the file name of the document, which includes the part name and material information. **Methods:** * \`ResetProgress()\`: Resets the progress handler. * \`UpdatePanels()\`: Updates the input and result panels based on the part's method and valid result. * \`SetNewPatternView()\`: Sets a new pattern view based on the part's working status. * \`SetPatternProperties()\`: Sets the pattern properties, such as boundary color, fill color, and thickness. **Events:** * \`OnDocumentPropertyChanged\`: Handles property changes in the document. * \`OnProgressHandlerPropertyChanged\`: Handles progress handler property changes. * \`OnPatternPropertyChanged\`: Handles pattern property changes. * \`OnMaterialPropertyChanged\`: Handles material property changes.**

Code File: SubPartStylePatternDocumentViewModel.cs

Summary:

SubPartStylePatternDocumentViewModel: The SubPartStylePatternDocumentViewModel class is responsible for managing the data and behavior of a sub-part style pattern document. It takes two inputs: * `IPart part`: The parent material part. * `FileController fileController`: A controller that handles file-related operations. This class provides several public properties and methods to facilitate the creation, updating, and display of panels related to the sub-part style pattern document. The main outputs of this class are: * `IVariantMaterial VariantMaterial`: The parent material variant. * `ObservableObject InputPanelViewModelMaterial` and `ObservableObject ResultPanelViewModelMaterial`: Two panels that display input and result data for the sub-part style pattern document. The class also includes several commands to execute tasks related to database lookups and panel updates. The `CalculatedValid` property indicates whether the calculations for the sub-part style pattern document are valid or not. Overall, the SubPartStylePatternDocumentViewModel class plays a crucial role in managing the data and behavior of sub-part style pattern documents within the application.

Code File: SubPatternCompoundDocumentViewModel.cs

Summary:

SubPatternCompoundDocumentViewModel: The SubPatternCompoundDocumentViewModel class is used to manage the properties and behavior of a compound document in the Procam Procost application. It provides a public interface for creating, modifying, and deleting compound documents. **Public Interface:** * **CreateCompoundCommand:** A command that creates a new compound document. * **DragOver:** A method that handles drag-and-drop operations to add or remove compound items from the document. * **Drop:** A method that adds or removes a compound item from the document when dropped onto it. **Properties:** * **Pattern:** The pattern associated with this document. * **Gid:** The unique identifier of the pattern. * **Style:** The style of the pattern. * **Title:** The title of the document, which is generated based on the pattern's name and modification status. * **FileName:** The file name of the document, which is generated based on the pattern's name and modification status. * **Patterns:** A collection of patterns associated with this document. * **Enlarge:** A property that controls the size of the compound items in the document. * **Gap:** A property that controls the gap between the compound items in the document. **Behavior:** * The class raises notifications when the pattern, style, or modification status changes. * It updates the title and file name based on the pattern's name and modification status. * It provides a command to create a new compound document. * It handles drag-and-drop operations to add or remove compound items from the document. **Dependencies:** * **IPattern:** The interface for patterns in the Procam Procost application. * **StyleDataInterfaces:** Interfaces for working with style data in the Procam Procost application.

Code File: SubPatternDocumentViewModel.cs

Summary:

Here is the technical summary: **SubPatternDocumentViewModel:** The SubPatternDocumentViewModel class provides a view model for sub-pattern documents in the application. It takes an IPattern object as input and allows users to access and manipulate pattern data. Public Interface: - Pattern: Retrieves or sets the underlying pattern data. - Gid: Retrieves the unique identifier of the pattern document. - Style: Retrieves the parent style

associated with the pattern. - StyleDataTypeName: Returns the name of the pattern data type. - Title: Retrieves the title of the pattern document, which is generated based on the pattern's name and modified status. - FileName: Retrieves the file name of the pattern document, which includes the pattern's name and parent style. This class also implements INotifyPropertyChanged to notify other parts of the application when the pattern data changes.

Code File: SubPatternHistoryDocumentViewModel.cs

Summary:

****SubPatternHistoryDocumentViewModel:** The SubPatternHistoryDocumentViewModel class manages the display of a sub-pattern history document. It takes an IPattern object as input and provides public properties for the pattern's name, modified status, and archived patterns. This class also maintains a list of archived patterns, sorted by their updated-by date. The list is used to generate a table view of the archived patterns, with each row representing a pattern and its corresponding header (e.g., "Archived No. 1"). The class raises property changed events when the pattern's modified status or name changes, which triggers updates to the document title and file name. Additionally, this class handles theme changes by updating the colors used for displaying patterns and their boundaries. It also disposes of any resources it manages when it is disposed.**

Code File: SubPatternsFormsDocumentViewModel.cs

Summary:

Here is the technical summary of the C# code: **SubPatternsFormsDocumentViewModel: The SubPatternsFormsDocumentViewModel class provides a view model for managing patterns and styles in an application. It takes an IStyleData object as input, which determines the type of style data being used. This view model exposes several properties, including: * PatternsList: A collection of pattern items * SelectedItems: A list of selected pattern items * GraphicMinWidth: The minimum width for graphical elements The class also provides a method to get the patterns list based on the input IStyleData object. Additionally, it has a property IsCalculateEnabled that is determined by the type of style data being used. Overall, this view model enables the management and manipulation of pattern lists in an application, providing a foundation for building user interfaces and handling user interactions.**

Code File: SubPatternsGridDocumentViewModel.cs

Summary:

Here is the technical summary: **SubPatternsGridDocumentViewModel: The SubPatternsGridDocumentViewModel class provides a view model for managing and displaying sub-patterns in a grid document. It takes two inputs: `IStyleData` and `IShellService`. This view model allows you to: * Display a list of patterns * Drag and drop patterns within the grid * Manage user-defined columns for customizing pattern display The class also provides properties for accessing the shell service, style data, and user-defined columns.**

Code File: SubPatternSplitDocumentViewModel.cs

Summary:

I'm an expert technical writer for a software company, and my goal is to generate documentation for the software. The code you provided appears to be part of a C# program that performs various geometric calculations, such as finding the intersection points of lines and circles. Here's a breakdown of the code: 1. `FindCircle` method: This method takes four points (two line segments) as input and returns the center, radius, start point, and end point of a circle that intersects with both line segments. 2. `LineEquation` method: This method takes two points as input and returns the coefficients (a, b, c) of the equation of a line in the form ax + by + c = 0. 3. `FindCrossing` method: This method takes four points (two line segments) as input and returns the point where the two lines intersect. 4. `LineCircleIntersection` method: This method takes three points (a circle center, radius, and a line segment) as input and returns an array of intersection points between the circle and the line segment. These methods seem to be used for geometric calculations in computer graphics or game development. As a technical writer, I would focus on documenting these methods and their usage scenarios. Here's a possible documentation outline: 1. Introduction 2. Method descriptions: * `FindCircle` * `LineEquation` * `FindCrossing` * `LineCircleIntersection` 3. Usage examples: * How to use the methods in a C# program * Example scenarios, such as finding the intersection points of two lines or a line and a circle 4. Troubleshooting tips: * Common issues that may arise when using these methods * Solutions to common problems Please let me know if you have any specific documentation requirements or preferences!

Code File: SubStyleDocumentViewModel.cs

Summary:

****SubStyleDocumentViewModel:** The SubStyleDocumentViewModel class is used to manage the data and behavior of a style document in the Procost application. It provides a public interface for interacting with the style document, including commands for creating and calculating graded sizes. **Public Interface:** * The class takes two inputs: + `IStyle` - the style document being managed + `FileController` - used to interact with file-related operations * The class exposes several properties that provide information about the style document, such as its name, modified status, and GID. * It also provides several commands for performing actions on the style document: + `CreateGradedSizesCommand`: creates graded sizes for a GradedStyle + `CalculateGradedSizesCommand`: calculates graded sizes for a GradedStyle + `MoreDbLookupsCommand` and `MoreDbOtherFieldsCommand`: perform database lookups and other field operations **Purpose:** The SubStyleDocumentViewModel class is designed to provide a centralized management point for style documents in the Procost application. It allows developers to interact with style documents using a set of commands and properties, making it easier to integrate style document functionality into the application.**

Code File: SubVariantDocumentViewModel.cs

Summary:

Here is the technical summary of the C# code: **SubVariantDocumentViewModel: The SubVariantDocumentViewModel class manages a sub-variant document in the application. It**

provides properties and commands to interact with the variant data, including its name, modified status, and working result. The class also handles property changes and disposes of resources when necessary. **Public Interface:** * **Variant**: Gets or sets the IVariant object representing the sub-variant. * **VariantResult**: Gets the valid result of the variant's working state. * **OriginalVariantResult**: Gets the original valid result of the variant, if available. * **IsModified**: Indicates whether the variant has been modified. * **Gid**: Gets or sets the unique identifier for the variant. * **Style**: Gets the parent style associated with the variant. * **StyleDataTypeName**: Returns the name of the data type for the variant (e.g., "Variant"). * **CalculatedValid**: Indicates whether the variant's working result is valid. * **IsValidationError**: Indicates whether there is an error in the variant's validation. * **Title**: Gets or sets the title of the sub-variant document, based on its name and modified status. * **FileName**: Gets or sets the file name of the sub-variant document. **Commands:** The class provides two commands: * **MoreDbLookupsCommand**: Executes a task to perform more database lookups. * **MoreDbOtherFieldsCommand**: Executes a task to perform more database operations for other fields.

Code File: SubVariantWizardDocumentViewModel.cs

Summary:

SubVariantWizardDocumentViewModel: The SubVariantWizardDocumentViewModel is a view model that manages the behavior of a sub-variant wizard document. It provides public interface for creating, applying, and canceling changes to the document's variants. **Public Interface:** * Creates new variants using the `NewVariantCommand`. * Applies all changes made to the variants using the `ApplyCommand`. * Cancels all changes made to the variants using the `CancelCommand`. **Properties:** * `Style`: The current style being edited. * `IsModified`: A flag indicating whether the document has been modified. * `Finished`: A flag indicating whether the wizard has finished (applied or canceled). * `Gid`: A unique identifier for the document. * `LocalWizardDetailView`: A flag controlling whether to display the wizard detail view. **Commands:** * `NewVariantCommand`: Creates a new variant and binds it to the data view. * `ApplyCommand`: Applies all changes made to the variants and marks the wizard as finished. * `CancelCommand`: Cancels all changes made to the variants and marks the wizard as finished.

Code File: UnderWorkViewModel.cs

Summary:

Here is the technical summary of the C# code: **UnderWorkViewModel:** The UnderWorkViewModel class is responsible for managing the data and behavior of the UnderWorkView. It provides a public interface to interact with the view, including properties and commands. **Properties:** * **SelectedItem:** Gets or sets the currently selected style data. * **SelectedItems:** Gets or sets the collection of selected items (styles). * **UnderWorkStyles:** Gets or sets the collection of underwork styles. * **IsPreviewModeOn:** Gets or sets whether the application is in preview mode. * **CurrentFilterText:** Gets or sets the current filter text. * **StyleDataEnumValues:** Gets the list of available style data enum values. **Commands:** The class provides several commands that can be used to interact with the view, including: * **FileService:** Provides access to file-related services. * **StyleOperationsService:** Provides access to style-related operations (e.g., creating, editing, deleting). * **ExportToReportService:** Provides access to exporting data to a report. * **ExportPictureService:** Provides access to exporting pictures. **Methods:** The class

includes several methods that can be used to initialize the view and manage its behavior, including: * **Initialize(): Initializes the view by setting up various properties and services. * **UpdateSelectedItems(IStyle parentStyle):** Updates the selected items collection when a style is closed (removes inner selected items). **Dispose():** Disposes of any resources used by the class. This summary focuses on what the code does for the application, rather than how it works.**

Code File: ViewsListViewModel.cs

Summary:

****ViewsListViewModel:** The ViewsListViewModel is responsible for managing the views list and providing a way to interact with it. It allows users to create, rename, restore, and delete views. **Public Interface:** * The ViewModel takes in several services as dependencies: + IShellService + IDocumentViewsService + IStyleOperationsService + ExportFactory + ExportFactory * It has four commands: + CreateNewViewCommand: Creates a new view and allows the user to enter a name. + SelectedViewCommand: Loads the selected view and makes it active. + RenameViewCommand: Renames an existing view. + DeleteViewCommand: Deletes an existing view. **Purpose:** The ViewsListViewModel is used to manage the views list, which is a collection of available views. It provides a way for users to interact with the views list by creating, renaming, restoring, and deleting views. The ViewModel also keeps track of the currently selected view and allows users to load it. **What it does:** * Manages the views list * Allows users to create new views * Allows users to rename existing views * Allows users to restore deleted views * Allows users to delete views * Keeps track of the currently selected view**

Code File: EditorViewModelBase.cs

Summary:

Here is the technical summary: **EditorViewModelBase: The EditorViewModelBase class represents a view-model for editing style data. It provides a public interface to interact with the editor, including setting and getting title, selected items, error message, and validation status. The class also includes commands for OK and Cancel actions, which can be used to close the editor. **Public Interface:** * Initializes the editor with a view and document views service * Sets and gets title, selected items, error message, and validation status * Provides OK and Cancel commands to close the editor **Purpose:** The EditorViewModelBase class is designed to manage the editing of style data. It provides a centralized location for handling user input, validating data, and displaying errors or success messages.**

Code File: MaterialEditorViewModel.cs

Summary:

Here is the technical summary: **MaterialEditorViewModel: The MaterialEditorViewModel class represents a view-model for editing style data documents. It provides a public interface to manage and display the selected method. **Public Interface:** * **SelectedMethod**: Gets or sets the currently selected method, allowing users to interact with the document editor. This view-model serves as a bridge between the user interface and the underlying application logic,**

enabling seamless interaction with style data documents.

Code File: PartEditorViewModel.cs

Summary:

Here is the technical summary: **PartEditorViewModel: The PartEditorViewModel class represents a view-model for editing part data. It provides a list of available types that can be used to filter and organize part data. **Public Interface:** * Initializes with an IPartEditorView and IDocumentViewsService * Returns a list of string values representing the available types This view-model is designed to facilitate the editing of part data, providing a way to access and manipulate the available types for filtering and organization.**

Code File: PatternEditorViewModel.cs

Summary:

Here is the technical summary: **PatternEditorViewModel: The PatternEditorViewModel class represents a view-model for editing pattern data. It provides a list of available types that can be used to edit patterns. **Public Interface:** * Initializes with an IPatternEditorView and IDocumentViewsService * Returns a list of available types (IList TypesList)**

Code File: StyleEditorViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **StyleEditorViewModel: The StyleEditorViewModel class represents the EditStyleData document view-model. It provides a way to manage and edit style data in your application. **Public Interface:** * Initializes with an IStyleEditorView and IDocumentViewsService * Allows you to interact with the view and document views service **Purpose:** This view-model enables you to create, read, update, and delete (CRUD) style data within your application. It provides a centralized management system for editing style data, making it easier to manage and maintain your application's style-related features.**

Code File: VariantEditorViewModel.cs

Summary:

Here is the technical summary: **VariantEditorViewModel: The VariantEditorViewModel class represents a document view-model for editing style data. It provides a public interface to interact with the application's editor and document views services. **Public Interface:** * Initializes a new instance of the VariantEditorViewModel class, requiring an IVariantEditorView and IDocumentViewsService as inputs. * Outputs a fully initialized VariantEditorViewModel object that can be used to manage editing style data.**

Code File: CustomizeDefaultsViewModel.cs

Summary:

This is a C# code for an expert technical writer for a software company. The goal is to generate documentation for various software components, including editors and style data types. The code defines several classes: 1. `StyleDataType`: This class represents a style data type, which can be one of the following: Style, Variant, VariantMaterial, or Part. It has properties for the name and fields collection. 2. `UserSelectableItem`: This class represents an item that can be selected by the user. It has properties for the name, extended name, and selection state. The code also defines several methods: 1. `GetSelectedStyleData(string name)`: This method returns a list of selected style data items based on the given name. 2. `GetSelectedResultStyleData(string name)`: This method returns a list of selected result style data items based on the given name. 3. `GetStyleDataVisibilityPropertyNames(Type styleDataType)`: This method returns a list of property names for the given style data type that are visible to the user. 4. `GetResultStyleDataVisibilityPropertyNames(Type styleDataType)`: This method returns a list of property names for the given result style data type that are visible to the user. 5. `AddIntoNamesList(IList namesList, Type styleDataType, Type visibilityFieldAttributeType)`: This method adds property names from the given style data type and attribute type into the names list. The code also defines several dictionaries and lists: 1. `StyleDataTypes`: A list of `StyleDataType` objects. 2. `ResultStyleDataTypes`: A list of `StyleDataType` objects for result style data types. 3. `CustomColors`: A dictionary that maps custom color names to their hex codes. The code uses the following frameworks and libraries: 1. .NET Framework 2. C# 7.0 3. LINQ (Language Integrated Query) 4. ObservableCollection (for observing changes in collections) Overall, this code is designed to generate documentation for various software components, including editors and style data types, based on user input and configuration settings.

Code File: CustomizeObjectAttribsViewModel.cs

Summary:

Here is the technical summary of the C# code: **CustomizeObjectsAttribsViewModel: The CustomizeObjectsAttribsViewModel class manages the options for customizing objects in a Procost application. It provides properties and commands to control the appearance and behavior of objects, including their themes, colors, and thickness. **Properties:** * `PreviewBackgroundColor`: A brush that represents the background color of the preview area. * `ResetableObjectThemes`: A dictionary of theme objects that can be applied to resetable base objects. * `SelectedResetableObjectTheme`: The currently selected theme object from the `ResetableObjectThemes` dictionary. * `SelectedObject`: The currently selected resetable base object. **Commands:** * `RestoreDefaultsCommand`: A command that restores the default settings for customizing objects. **Methods:** * `Initialize()`: Initializes the view model by loading the default settings and selecting a theme. * `SaveSettings()`: Saves the current settings to the ObjectTypesSettings instance. * `UpdateObjectTypesWithProperties()`: Updates the object types with properties from an ObjectTypesSettings instance. * `ChangePreviewBackgroundColor()`: Changes the preview background color based on the selected theme. This summary focuses on what the code does for the application, rather than how it works. It provides a concise and end-user-friendly overview of the class's purpose, properties, commands, and methods.**

Code File: CustomizeQatViewModel.cs

Summary:

Here is the technical summary of the C# code: **CustomizeQatViewModel: The CustomizeQatViewModel class manages the customization options for a Quick Access Toolbar (QAT) in an application. It provides public properties and commands to interact with the QAT, including adding, removing, moving up, and moving down actions. **Public Interface:** * Initializes the view model with ribbon service data * Provides observable collections of tab models and QAT command models * Allows selection of a tab or command from a list box * Offers commands for adding, removing, moving up, and moving down items in the QAT **Key Features:** * Manages the selection of tabs and commands in the QAT * Enables adding and removing actions from the QAT * Provides commands to move up and down actions in the QAT * Raises can-execute changed events for commands that depend on the selected item**

Code File: CustomizeRibbonViewModel.cs

Summary:

Here is the technical summary of the C# code: **CustomizeRibbonViewModel: The CustomizeRibbonViewModel class manages the customization options for a ribbon in an application. It provides a public interface to interact with the ribbon, including creating, renaming, and deleting tabs and groups. **Public Interface:** * Initialize method: Initializes the view model with default tab models and custom tab models. * TabModelsFrom and TabModelsTo properties: Get/Set collections of TabModel objects representing the from and to tabs. * SelectedTabFrom and SelectedTabTo properties: Get/Set TabModel objects representing the selected tabs. * CreateTab, DeleteTab, RenameTab, CreateGroup, DeleteGroup, RenameGroup, AddToRibbon, RemoveFromRibbon, MoveUp, and MoveDown commands: Execute actions related to creating, renaming, deleting, and moving tabs and groups. **Purpose:** The CustomizeRibbonViewModel class enables users to customize the ribbon in an application by creating, renaming, and deleting tabs and groups. It provides a set of commands that can be executed to perform these actions.**

Code File: CustomizeShortcutKeysViewModel.cs

Summary:

Here is the technical summary: **CustomizeShortcutKeysViewModel: The CustomizeShortcutKeysViewModel class manages the customization of shortcut keys for various actions in a software application. It provides a public interface to initialize and save settings related to shortcut keys. **Public Interface:** * Initialize(IRibbonService ribbonService): Initializes the view model by populating the list of available tabs, setting default shortcuts, and updating the selected tab. * SaveSettings(): Saves the current shortcut key settings for each action, and updates the input bindings accordingly. **Purpose:** The CustomizeShortcutKeysViewModel class enables users to customize the shortcut keys for various actions in the application. It provides a centralized management system for storing and retrieving shortcut key settings, allowing users to personalize their workflow.**

Code File: OptionsWindowModel.cs

Summary:

Here is the technical summary: **OptionsWindowModel: The OptionsWindowModel class manages the options window for a software application. It provides a public interface to display and configure various options. **Public Interface:** * Displays a list of options, including "CustomizeRibbon", "QuickAccessToolbar", "CustomizeUserDefaults", "CustomizeObjectsAttribs", and "CustomizeShortcutKeys". * Allows the user to select an option from the list. * Provides commands for OK and Cancel actions. * Enables the application to show or close the options window. **Functionality:** The OptionsWindowModel class handles the display of the options window, including populating the list of options and handling user interactions. It also provides methods for showing and closing the window, as well as tracking whether the user has cancelled the operation.**

Code File: MaterialInputALP_LE6PanelViewModel.cs

Summary:

Here is the technical summary: MaterialInputALP_LE6PanelViewModel: The MaterialInputALP_LE6PanelViewModel class provides a view model for managing material input data related to ALP LE6 panels. This class serves as a bridge between the user interface and the underlying business logic, allowing developers to create applications that interact with this type of panel. Public Interface: This class does not have any specific inputs or outputs; instead, it inherits its functionality from the MaterialInputCommonPanelViewModel base class.

Code File: MaterialInputALS_LEAPanelViewModel.cs

Summary:

****MaterialInputALS_LEAPanelViewModel:** The MaterialInputALS_LEAPanelViewModel is a view model that provides data and functionality for the ALS LEA panel in the Procost application. It inherits from the MaterialInputCommonPanelViewModel, allowing it to share common properties and methods with other material input panels. This view model does not have any specific inputs or outputs, but rather serves as a container for data and logic related to the ALS LEA panel.**

Code File: MaterialInputBasicPanelViewModel.cs

Summary:

****MaterialInputBasicPanelViewModel:** The MaterialInputBasicPanelViewModel is a view model that provides the necessary data and functionality for displaying basic material input information in an application. This view model serves as a foundation for creating panels that allow users to input and manage material-related data. It inherits its common properties and behaviors from the MaterialInputCommonPanelViewModel class, making it a versatile tool for building various material input interfaces.**

Code File: MaterialInputCombinedPartLayoutPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialInputCombinedPartLayoutPanelViewModel: This view model provides a combined panel layout for material input and part information. It allows users to manage and display both material input data and part details in a single panel. The public interface enables you to interact with this view model, making it suitable for use in your application's user interface.

Code File: MaterialInputCommonPanelViewModel.cs

Summary:

Here is the technical summary: **MaterialInputCommonPanelViewModel: The MaterialInputCommonPanelViewModel class provides a way to manage and display material information for an application. It exposes a public property called VariantMaterial, which allows users to set or retrieve variant material data. Public Interface: - Input: VariantMaterial (IVariantMaterial interface) - Output: None This class is designed to facilitate the integration of material-related data into the application's user interface, making it easier for end-users to access and manipulate this information.**

Code File: MaterialInputManualPositionRowsPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **MaterialInputManualPositionRowsPanelViewModel: This class provides a view model for managing manual position rows in material input applications. It allows users to interact with and manipulate data related to manual position rows, serving as an interface between the application's user interface and its underlying logic.**

Code File: MaterialInputOLNPanelViewModel.cs

Summary:

****MaterialInputOLNPanelViewModel:** The MaterialInputOLNPanelViewModel is a view model that provides data and functionality for the OLN (Other Labor and Non-Labor) material input panel in the Procost application. This view model inherits from the MaterialInputCommonPanelViewModel, allowing it to share common properties and methods with other material input panels. **Public Interface:** * The view model does not have any specific inputs or outputs, as its primary purpose is to provide a framework for managing data and behavior related to OLN material input.**

Code File: MaterialInputPACPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialInputPACPanelViewModel: The MaterialInputPACPanelViewModel provides a view model for managing material input data in the Procost application. This class serves as a foundation for creating panels that handle material input operations.

Code File: MaterialInputTXN_VPNPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialInputTXN_VPNPanelViewModel: The MaterialInputTXN_VPNPanelViewModel provides a view model for managing material input transactions and VPN panel data. This class serves as a foundation for creating and manipulating material input transactions, allowing users to interact with relevant data and functionality within the application.

Code File: MaterialInput_AtWork_PanelViewModel.cs

Summary:

****MaterialInput_AtWork_PanelViewModel:** The MaterialInput_AtWork_PanelViewModel is a view model that provides data and functionality for the Material Input at Work panel in the Procost application. This view model inherits from MaterialInputCommonPanelViewModel, allowing it to share common properties and methods with other material input panels. Public Interface: * The view model does not have any specific inputs or outputs, as its primary purpose is to provide a foundation for displaying and managing material input data within the panel. In summary, this view model enables the Procost application to display and manage material input data in the Material Input at Work panel.**

Code File: MaterialResultALPPanelViewModel.cs

Summary:

****MaterialResultALPPanelViewModel:** The MaterialResultALPPanelViewModel is a view model that manages the display and behavior of an ALP (Advanced Life Prediction) panel in the Procost application. This class provides a public interface for interacting with the panel, allowing users to access and manipulate its contents. Public Interface: * No inputs or outputs are explicitly defined, as this view model primarily focuses on managing the panel's state and behavior rather than processing external data.**

Code File: MaterialResultALSPanelViewModel.cs

Summary:

MaterialResultALSPanelViewModel: The MaterialResultALSPanelViewModel is a view model that provides data and functionality for the Material Result ALS panel in the Procost application.

It inherits from the MaterialResultCommonPanelViewModel, allowing it to share common properties and methods with other material result panels. Public Interface: This view model does not have any specific inputs or outputs, as its primary purpose is to provide a framework for displaying and managing data related to material results.

Code File: MaterialResultBasicPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResultBasicPanelViewModel: The MaterialResultBasicPanelViewModel provides a basic panel view model for displaying material results. It inherits from the MaterialResultCommonPanelViewModel and offers a straightforward interface for presenting material-related data to users.

Code File: MaterialResultCommonPanelViewModel.cs

Summary:

Here is the technical summary: **MaterialResultCommonPanelViewModel: The MaterialResultCommonPanelViewModel class provides a view model for managing material results in an application. It allows users to access and manipulate material result data. **Public Interface:** * The class accepts no inputs. * It exposes two properties: + `MaterialResult`: Retrieves or sets the current material result data. + `OriginalMaterialResult`: Retrieves or sets the original material result data. This view model enables applications to work with material results, making it a crucial component for managing and processing this type of data.**

Code File: MaterialResultLE6PanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResultLE6PanelViewModel: The MaterialResultLE6PanelViewModel provides a view model for displaying material result data in an LE6 panel. This class serves as a foundation for building a user interface that presents material results to users.

Code File: MaterialResultLEAPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResultLEAPanelViewModel: The MaterialResultLEAPanelViewModel provides a view model for displaying material results in the LEA (Labor Estimating and Accounting) panel. This class serves as a foundation for managing data related to material costs and estimates, allowing users to access and manipulate relevant information within the application.

Code File: MaterialResultManualPositionRowsPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResultManualPositionRowsPanelViewModel: This class provides a view model for managing manual position rows in material result panels. It allows users to interact with and manipulate data related to manual position rows, enabling them to perform tasks such as viewing, editing, or deleting this type of data.

Code File: MaterialResultOLNPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResultOLNPanelViewModel: The MaterialResultOLNPanelViewModel provides a view model for managing material results in an OLN (Optical Lens Network) panel. This class serves as a foundation for displaying and manipulating material-related data within the application.

Code File: MaterialResultTXN_VPNPanelViewModel.cs

Summary:

****MaterialResultTXN_VPNPanelViewModel:** The MaterialResultTXN_VPNPanelViewModel is a view model that provides data and functionality for the VPN panel in the Procost application. It inherits from the MaterialResultCommonPanelViewModel, allowing it to share common properties and methods with other panels. This view model serves as an intermediary between the user interface and the underlying business logic, enabling the display of relevant information and handling of user interactions related to the VPN panel.**

Code File: MaterialResult_AtWork_PanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: MaterialResult_AtWork_PanelViewModel: This class provides a view model for the material result at work panel. It allows you to manage and display material results in your application. The public interface includes inputs and outputs that enable you to interact with the panel and retrieve relevant information.

Code File: OtherPartResultsPanelViewModel.cs

Summary:

Here is the technical summary: **OtherPartResultsPanelViewModel: The OtherPartResultsPanelViewModel class provides a view model for displaying and managing**

part results. It allows users to set an IPart object, which populates a collection of IPartResult objects based on the part's working material. The view model also exposes a Results property that returns a filtered and sorted list of part results based on their calculation source and efficiency. **Public Interface: * Set an IPart object using the Part property * Get a filtered and sorted list of part results using the Results property**

Code File: PatternInputALPPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputALPPanelViewModel: The PatternInputALPPanelViewModel provides a view model for managing pattern input data related to ALP (Automated Laser Processing) panels. This class serves as a foundation for displaying and processing panel-specific information within the application, allowing users to interact with and manipulate relevant data.**

Code File: PatternInputALSPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputALSPanelViewModel: The PatternInputALSPanelViewModel provides a view model for managing pattern input data related to ALS (Automated Laser Scanning) panels. This class serves as a foundation for building user interfaces that interact with ALS panel data, allowing developers to create applications that efficiently handle and process this type of data.**

Code File: PatternInputATMPanelViewModel.cs

Summary:

****PatternInputATMPanelViewModel:** The PatternInputATMPanelViewModel is a view model that provides the necessary data and functionality for managing pattern input related to ATM (Automated Teller Machine) operations. This view model serves as a bridge between the user interface and the underlying business logic, allowing developers to create a seamless and intuitive user experience. **Public Interface:** This view model does not have any specific inputs or outputs, but rather inherits its functionality from its parent class PatternInputCommonPanelViewModel. As such, it provides a common set of features and data for managing pattern input operations across various applications.**

Code File: PatternInputBasicPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputBasicPanelViewModel: The PatternInputBasicPanelViewModel provides a basic panel view model for pattern input applications. This class serves as a foundation for creating panels that allow users to input patterns in Procam's Procost application.**

Code File: PatternInputCommonPanelViewModel.cs

Summary:

Here is the technical summary: **PatternInputCommonPanelViewModel: The PatternInputCommonPanelViewModel class manages the data and behavior of a panel that displays and edits pattern input settings. It provides public properties for accessing and modifying the part, underwork part, predefined rotation angles, and predefined rotation angle. **Public Interface:** * Part: Gets or sets the current part. * UnderworkPart: Gets or sets the underwork part, which updates the Part property accordingly. * PredefinedRotationAngles: A list of predefined rotation angles. * PredefinedRotationAngle: Gets or sets the current predefined rotation angle, which is updated based on changes to the Part property. **Behavior:** The class also implements INotifyPropertyChanged and IDisposable interfaces. It raises PropertyChanged events when the Part property changes, and it disposes of resources when the Dispose method is called.**

Code File: PatternInputManualPositionRowsPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputManualPositionRowsPanelViewModel: The PatternInputManualPositionRowsPanelViewModel provides a view model for managing manual position rows in a pattern input application. This class serves as a foundation for creating and manipulating manual position rows, allowing users to interact with them in a controlled manner. Public Interface: This view model does not have any specific inputs or outputs; instead, it focuses on providing a framework for working with manual position rows within the application.**

Code File: PatternInputMSLPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputMSLPanelViewModel: The PatternInputMSLPanelViewModel provides a view model for managing pattern input data related to MSL (Material Safety Label) panels in Procost applications. This class serves as a foundation for creating and manipulating MSL panel views, allowing developers to integrate this functionality into their applications. **Public Interface:** This view model does not have any specific inputs or outputs; instead, it provides a framework for managing MSL panel data within the application.**

Code File: PatternInputSRCPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternInputSRCPanelViewModel: The PatternInputSRCPanelViewModel provides a view**

model for managing pattern input data in the Procost application. This class serves as a foundation for creating panels that handle pattern input operations.

Code File: PatternInputTXN_VPNPanelViewModel.cs

Summary:

Here is the technical summary: **PatternInputTXN_VPNPanelViewModel: The PatternInputTXN_VPNPanelViewModel class manages the calculation modes and maximum results for a pattern input panel. It provides a public interface to set and retrieve the calculation mode and maximum number of results. **Public Interface:** * `CalcModes`: A dictionary that maps calculation modes (e.g., AUTO, General, I row) to their corresponding names. * `IsIlock`: A boolean indicating whether the calculation mode is locked or not. * `MaxResultsList`: An enumerable list of integers representing the maximum number of results for each calculation mode. * `MaxResultsCount`: An integer property that sets and retrieves the maximum number of results. **Purpose:** The PatternInputTXN_VPNPanelViewModel class updates its calculation modes and maximum results based on changes to the part material method, rectangle sheet calculation mode, or other relevant properties. It also provides a way to set and retrieve the maximum number of results for each calculation mode.**

Code File: PatternInput_AtWork_PanelViewModel.cs

Summary:

Here is the technical summary: **PatternInput_AtWork_PanelViewModel: The PatternInput_AtWork_PanelViewModel provides a view model for managing pattern input data in an At-Work panel application. This class exposes a single property, **Part**, which represents a part object that contains relevant information about a specific pattern. Public Interface: * Part (IPart): Retrieves or sets the part object associated with this view model.**

Code File: PatternResultALS_ALPPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternResultALS_ALPPanelViewModel: The PatternResultALS_ALPPanelViewModel class provides a view model for displaying and managing pattern results in the Procost application. This class inherits from the PatternResultCommonPanelViewModel, allowing it to leverage common functionality while still offering its own unique features. As an end-user, you can expect this view model to enable you to interact with pattern results in a meaningful way, potentially including tasks such as viewing, editing, and analyzing patterns.**

Code File: PatternResultATMPanelViewModel.cs

Summary:

Here is the technical summary: **PatternResultATMPanelViewModel: The PatternResultATMPanelViewModel class provides a view model for an ATM (Automatic Teller Machine) panel in a pattern result application. It retrieves and exposes two key values: ANGLE_POS and SEMI_POS, which represent the angle position and semi-position of a part result, respectively. **Public Interface:** * ANGLE_POS: A string value representing the angle position of a part result. * SEMI_POS: A string value representing the semi-position of a part result.**

Code File: PatternResultBasicPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternResultBasicPanelViewModel: The PatternResultBasicPanelViewModel provides a basic panel view model for pattern results. It allows developers to create and manage panels that display pattern result data in their applications. This class serves as a foundation for building custom panels with specific functionality, offering a starting point for creating unique user interfaces.**

Code File: PatternResultCommonPanelViewModel.cs

Summary:

Here is the technical summary: **PatternResultCommonPanelViewModel: The PatternResultCommonPanelViewModel class provides a view model for managing pattern results in an application. It allows users to access and manipulate two types of part results: the current part result and the original part result. **Public Interface:** * Users can set or get the current part result using the `PartResult` property. * Users can set or get the original part result using the `OriginalPartResult` property.**

Code File: PatternResultMSLPanelViewModel.cs

Summary:

Here is a concise, end-user-friendly technical summary of the C# code: **PatternResultMSLPanelViewModel: The PatternResultMSLPanelViewModel provides a view model for managing pattern results in the Procost application. This class serves as a foundation for displaying and processing pattern-related data, offering a public interface for interacting with pattern result information.**

Code File: PatternResultSRCPanelViewModel.cs

Summary:

****PatternResultSRCPanelViewModel:** The PatternResultSRCPanelViewModel is a view model that provides data and functionality for the Pattern Result Source Panel application. This view model serves as a bridge between the user interface and the underlying business logic,**

allowing developers to create a robust and maintainable application. Public Interface: * The view model does not have any specific inputs or outputs, but it inherits properties and methods from its base class, PatternResultCommonPanelViewModel.

Code File: PatternResultTXN_VPNPanelViewModel.cs

Summary:

Here is the technical summary: **PatternResultTXN_VPNPanelViewModel: The PatternResultTXN_VPNPanelViewModel class provides a view model for displaying pattern result transaction VPN panel data. It exposes a single property, CalculationMode, which returns the calculation mode of the part result as a string. The input to this property is the part result object, and it returns an empty string if no part result is available.**

Code File: PatternResult_AtWork_PanelViewModel.cs

Summary:

Here is the technical summary: **PatternResult_AtWork_PanelViewModel: The PatternResult_AtWork_PanelViewModel class provides a view model for managing data related to pattern results at work. This class allows you to get and set information about material results. Public Interface: * No inputs or outputs are explicitly defined, as this class primarily serves as a container for managing data. In summary, the PatternResult_AtWork_PanelViewModel class enables you to interact with and manipulate data related to pattern results at work within your application.**

Code File: IEditorViewBase.cs

Summary:

****IEditorViewBase:** The IEditorViewBase interface provides a set of methods for managing the display and closure of editor views in an application. This includes showing a dialog box with an optional owner object, as well as closing the view. Public Interface: * `ShowDialog(object owner)`: Displays a dialog box with an optional owner object. * `Close()`: Closes the editor view.**