**Multi-Functional GUI Application Report**

**Introduction**

This report presents a comprehensive overview of a multi-functional GUI application developed using Java Swing. The application facilitates the management of hierarchical data, attributes, and groupings through an intuitive graphical user interface. The application's architecture and components are explained, demonstrating their roles in achieving the application's functionality.

**Table of Contents**

1. **Application Overview**
2. **Components and Classes**
   * **JCheckBoxTree**: Custom Checkbox Tree Component
   * **CustomTreeCellRenderer**: Custom Tree Cell Renderer
   * **Groups**: Panel for Managing Node Groups
   * **AttributesTableModel**: Table Model for Node Attributes
   * **UniqueAttributesManager**: Singleton Manager for Unique Attributes
   * **ProjectWindow**: Main GUI Panel for the Application
   * **Internal**: Main Class to Launch Internal Frames

**1. Application Overview**

The application is designed to manage hierarchical data structures with associated attributes and groupings. Users can visualize and interact with the data through a GUI, making it easy to organize and manipulate complex data hierarchies.

**2. Components and Classes**

**1. JCheckBoxTree**: Custom Checkbox Tree Component

The **JCheckBoxTree** class is a Java class that extends the **JTree** class from the Swing framework. It introduces a customizable tree component with enhanced features, particularly related to handling checkboxes for tree nodes. The class allows users to select and manage nodes within the tree by checking or unchecking checkboxes associated with each node.

**Class Overview:**

The **JCheckBoxTree** class extends the **JTree** class and adds a range of functionalities for managing the state of checkboxes associated with tree nodes. The primary features of this class include:

1. **Checkbox Management:** The central feature of the **JCheckBoxTree** class is the ability to associate checkboxes with individual nodes in the tree. This is achieved by using a custom cell renderer that displays checkboxes based on the state of each node. The class provides mechanisms to manage the selection state of nodes through checkboxes.
2. **CheckedNode Data Structure:** The class introduces a **CheckedNode** class that holds information about the selection state of a node. This includes whether the node is selected, whether it has children, and whether all its children are selected. This data structure is used to efficiently manage and track the state of checkboxes for nodes.
3. **Event Handling:** The class defines a new event type called **CheckChangeEvent**, along with the necessary event listeners and handling methods. This enables developers to respond to changes in checkbox selection states within the tree. Event listeners can be registered to receive notifications when checkbox states change.
4. **Selection Model Override:** The **JCheckBoxTree** class overrides the default selection model of the **JTree** to control the selection behavior. It replaces the conventional selection mechanism with a custom mechanism based on checkbox states.
5. **Attribute Management:** The class provides methods for associating attributes with individual nodes. It allows developers to attach key-value pairs of attributes to nodes and retrieve them later. This could be useful for storing metadata or additional information about tree nodes.

**Usage:**

Developers can use the **JCheckBoxTree** class to create tree components with checkboxes for node selection. Key steps include:

1. **Creating an Instance:** Instantiate the **JCheckBoxTree** class, providing it with a **DefaultMutableTreeNode** as the root node. This will initialize the tree with the given root node.
2. **Checkbox Management:** The class handles checkbox management automatically based on the structure of the tree and user interactions. Nodes can be selected or deselected by clicking their associated checkboxes.
3. **Event Handling:** Developers can add listeners to the **JCheckBoxTree** instance to receive notifications when checkbox states change. This can be achieved using the **addCheckChangeEventListener** method.
4. **Attribute Management:** The class allows attaching attributes to nodes using the **setAttributeForNode** method. Attributes can then be retrieved using methods like **getAttributeForNode** and **getAllAttributesForNode**.
5. **Customization:** The appearance and behavior of the checkboxes and nodes can be customized using the provided methods and through overriding the provided cell renderer.

**Conclusion:**

The **JCheckBoxTree** class enhances the capabilities of the Swing framework's **JTree** component by providing a mechanism to manage node selection using checkboxes. This class streamlines the process of working with tree structures that require complex selection interactions, making it particularly useful for applications that involve hierarchical data representation and management.

1. **CustomTreeCellRenderer**: Custom Tree Cell Renderer

The **CustomTreeCellRenderer** class is a Java class that extends the **DefaultTreeCellRenderer** class, which is part of the Swing framework for building graphical user interfaces (GUIs). This custom renderer is intended to be used with a **JTree** component to customize the way tree nodes are displayed within the tree structure. The primary purpose of this class is to provide a means of associating specific icons with different nodes in the tree based on certain conditions.

**Class Overview:**

The **CustomTreeCellRenderer** class overrides the **getTreeCellRendererComponent** method of the parent **DefaultTreeCellRenderer** class. This method is responsible for rendering individual cells in a JTree component. By extending this class and customizing the rendering behavior, developers can display nodes in the tree with icons tailored to the data or context represented by those nodes.

**Key Features:**

1. **Icon Customization:** The core feature of the **CustomTreeCellRenderer** class is its ability to set custom icons for nodes in the JTree. The class achieves this by determining the appropriate icon based on the **userObject** associated with each node. The **userObject** can be any arbitrary object representing the data held by the node. In the provided code, there's an example condition where the **userObject** is checked against the string "CARE". If the condition is met, a custom icon (specified as "CARElogo.png") is associated with that node.
2. **Scaling Icons:** To ensure consistent icon sizes, the class provides a utility method called **getScaledIcon**. This method takes an **ImageIcon** and scales it to the desired width and height while maintaining the image's aspect ratio. This is important to maintain a visually pleasing appearance when icons of varying sizes are displayed in the tree.

**Usage:**

To utilize the **CustomTreeCellRenderer** class, developers would follow these steps:

1. Create an instance of the **CustomTreeCellRenderer** class.
2. Assign the instance to the **setCellRenderer** method of the **JTree** component for which you want to customize the rendering.

**Note:** It's important to ensure that the icons referenced in the class constructor or elsewhere in the code are located in the correct path relative to the running application.

**Conclusion:**

The **CustomTreeCellRenderer** class offers a powerful way to customize the appearance of nodes within a **JTree** component by associating specific icons with different nodes based on their data. By extending the capabilities of the **DefaultTreeCellRenderer** class, developers can create visually appealing and contextually relevant representations of hierarchical data structures.

1. **Groups**: Panel for Managing Node Groups

The **Groups** class, residing within the **com.mycompany.hmmm** package, is a crucial element of the multi-functional GUI application. This class is designed to manage and visualize groups of nodes within a hierarchical structure through a user-friendly graphical interface. This report provides an in-depth analysis of the **Groups** class, detailing its composition, functionalities, and its significance within the broader application context.

*Class Overview:*

The primary purpose of the **Groups** class is to offer the following functionalities:

1. **Group Management:** Facilitates the creation and management of groups composed of related nodes.
2. **Hierarchical Visualization:** Displays the hierarchical structure of groups and nodes through a custom **JTree** component.
3. **User Interaction:** Utilizes the **JOptionPane** for user input when adding new groups and nodes.

*Key Components:*

**Fields:**

* **groupsMap**: A **Map** that holds the relationships between group names and lists of associated node names.
* **root**: Represents the root node of the **JTree** structure, acting as the starting point for displaying groups and nodes.
* **groupsTree**: An instance of the **JTree** class that presents the hierarchical view of groups and nodes.
* **noGroupsLabel**: A **JLabel** used to convey a "No groups yet" message when no groups have been added.

**Constructor:**

* **Groups()**: Initializes the visual components of the class, setting the background color and layout. It also initializes the **groupsMap**, creates the root node, initializes the **groupsTree**, and configures the **noGroupsLabel**.

**Methods:**

1. **addNodes(List<String> nodes)**: Enables users to establish a group by specifying a list of associated nodes. It prompts users for a group name using a **JOptionPane** input dialog. If a valid group name is provided, it creates a new group node, associates the specified nodes with the group, and updates the **JTree** and **groupsMap** accordingly. The tree structure is then refreshed using the **reload()** method from the **DefaultTreeModel**.
2. **hasGroups()**: Provides a utility function to check if there are any groups present in the **groupsMap**.

**Functionality:**

1. **Creating Groups and Nodes:** Users can define new groups and associate them with specific nodes. The user is prompted to input a group name and select nodes for inclusion in the group. The group and its associated nodes are integrated into the **groupsMap**.
2. **Hierarchical Display:** The class uses a customized **JTree** component to visually represent the created groups and nodes. The root node serves as the parent for groups, and each group node contains its associated nodes.
3. **Updating the Display:** Upon adding new groups or nodes, the **JTree** structure is updated using the **reload()** method from **DefaultTreeModel**, ensuring the display reflects the most recent changes.
4. **User Feedback:** To enhance user feedback, a label is displayed with the message "No groups yet" when no groups are present. Once a group is added, this label is removed to provide a more intuitive user experience.

**Conclusion:**

In conclusion, the **Groups** class plays a pivotal role in the multi-functional GUI application, enabling efficient organization and visualization of groups and nodes. Its intuitive user interface, hierarchical presentation, and user feedback mechanisms collectively contribute to an enhanced user experience. This class exemplifies the skillful implementation of Java Swing components to realize essential functionalities within the application's framework.

**4. AttributesTableModel**: Table Model for Node Attributes

The **AttributesTableModel** class in the given code snippet is a Java class that extends **AbstractTableModel**. It's designed to represent and manage a table model for attributes associated with specific nodes. This class facilitates the display and management of attribute data in a tabular format, with two columns: "Attribute Name" and "Attribute Value".

**Class Overview:**

The **AttributesTableModel** class provides functionality for managing and displaying attribute data in a tabular format. The main features of this class include:

1. **Attribute Data Management:** The class manages attribute data using a map called **nodeAttributes**, which associates node names with lists of attribute names. It also maintains two separate lists: **attributeNames** and **attributeValues**, which store the names and values of attributes.
2. **Setting Node Attributes:** The **setNodeAttributes** method adds an attribute to the specified node's attribute list. It also updates the **attributeNames** and **attributeValues** lists accordingly and notifies the table model that the data has changed using **fireTableDataChanged()**.
3. **Clearing Node Attributes:** The **clearNodeAttributes** method removes all attributes associated with a given node. It clears the node's attribute list, as well as the **attributeNames** and **attributeValues** lists.
4. **Checking for Attributes:** The **hasAttribute** method checks if a given attribute name and value pair exist in the model. It iterates through the lists of attribute names and values to determine if a match is found.
5. **Table Model Implementation:** The class extends **AbstractTableModel**, which is the basis for implementing table models in Swing applications. It provides methods required by the table model interface, such as **getRowCount**, **getColumnCount**, **getValueAt**, and **getColumnName**.
6. **Displaying Table:** The **displayTable** method opens a new **JFrame** to display the attribute data in a JTable. If no attributes are associated with the specified node, it shows a warning message using **JOptionPane**.
7. **Table Data Retrieval:** The **getTableAsMatrix** method retrieves the attribute data as a two-dimensional array (matrix) of strings. Each row of the matrix contains an attribute name and its corresponding value.

**Usage:**

Developers can use the **AttributesTableModel** class in the following manner:

1. **Instantiation:** Create an instance of the **AttributesTableModel** class to manage attribute data for nodes.
2. **Attribute Management:** Use the **setNodeAttributes** method to add attributes to nodes. Call **clearNodeAttributes** to remove attributes associated with a specific node.
3. **Checking for Attributes:** Utilize the **hasAttribute** method to check if a given attribute name and value pair exist in the model.
4. **Table Display:** To display the attributes in a table, call the **displayTable** method with the node name as an argument.
5. **Table Data Retrieval:** Call the **getTableAsMatrix** method to obtain attribute data as a matrix for further processing or display.

**Conclusion:**

The **AttributesTableModel** class serves as a key component for managing and displaying attribute data associated with nodes in a tabular format. By extending **AbstractTableModel**, it adheres to the Swing framework's standards for implementing table models, making it a versatile and adaptable solution for displaying node attribute data.

**5. UniqueAttributesManager**: Singleton Manager for Unique Attributes

The **UniqueAttributesManager** class is a Java class designed to manage and display a list of unique attribute names. This class implements the singleton design pattern to ensure that there is only one instance of the **UniqueAttributesManager** class throughout the application's lifecycle. The primary purpose of this class is to keep track of unique attribute names and provide methods to display them in a user-friendly way.

**Class Overview:**

The **UniqueAttributesManager** class provides functionality for managing unique attribute names. The key features of this class include:

1. **Singleton Design Pattern:** The class follows the singleton pattern, which ensures that only one instance of **UniqueAttributesManager** exists within the application. This is achieved through a private constructor and a static method **getInstance()**.
2. **Unique Attribute Names List:** The class maintains a list called **uniqueAttributeNames**, which stores unique attribute names that have been added. The list is initialized in the constructor.
3. **Adding Unique Attribute Names:** The **addUniqueAttributeName** method is used to add a new unique attribute name to the list. If the attribute name is not already in the list, it's added. Otherwise, a message indicating that the attribute already exists is displayed.
4. **Getting Unique Attribute Names:** The **getUniqueAttributeNames** method returns the list of unique attribute names.
5. **Displaying Unique Attributes:** The **displayUniqueAtts** method creates a graphical display of the unique attribute names using a **JTable**. It creates a **JFrame** to host the **JTable** within a **JScrollPane** for scrolling if needed.

**Usage:**

Developers can use the **UniqueAttributesManager** class in the following way:

1. **Getting an Instance:** Use the **getInstance** method to obtain an instance of the **UniqueAttributesManager** class. This ensures that there is only one instance throughout the application.
2. **Adding Unique Attribute Names:** Call the **addUniqueAttributeName** method to add unique attribute names to the list.
3. **Retrieving Unique Attribute Names:** Use the **getUniqueAttributeNames** method to retrieve the list of unique attribute names.
4. **Displaying Unique Attributes:** Call the **displayUniqueAtts** method to display the unique attribute names in a graphical interface.

**Conclusion:**

The **UniqueAttributesManager** class is a valuable component for managing and displaying unique attribute names in a graphical user interface. By implementing the singleton pattern, it ensures that unique attribute names are consistently tracked and can be conveniently displayed when needed.

**6. ProjectWindow**: Main GUI Panel for the Application

The **ProjectWindow** class represents the main user interface for managing and interacting with a tree structure of nodes. It provides various functionalities to add, delete, group, and manage attributes associated with nodes in the tree. This class utilizes several components, including a **JCheckBoxTree**, a **JTextArea**, and buttons for various actions.

**Class Overview:**

The **ProjectWindow** class serves as the central component of the user interface. It includes the following key features:

1. **Layout and Components:** The class sets up a graphical user interface using the **JPanel** as its base component. It includes multiple panels for different sections, such as the tree panel, edit panel, data panel, and groups panel. These panels are arranged using the **GridBagLayout** and **GridLayout** managers.
2. **Tree Panel:** The tree panel displays the hierarchical structure of nodes using a **JCheckBoxTree**. The **JCheckBoxTree** is customized with a cell renderer to display checkboxes next to each node. The tree panel also uses a custom font for better readability.
3. **Data Panel:** The data panel includes a **JTextArea** for displaying information about the selected node, its parent, and grandparent nodes. The information is updated whenever the user selects a different node in the tree.
4. **Edit Panel:** The edit panel contains buttons for various actions, such as adding nodes, deleting nodes, grouping nodes, adding attributes, displaying attributes, displaying unique attributes, selecting all nodes, and selecting nodes based on attributes.
5. **Attributes Management:** The class maintains a mapping from nodes to their associated **AttributesTableModel** instances using the **nodeToAttributesTableModel** map. This allows the class to manage and display attributes associated with nodes.
6. **Event Listeners:** The class includes event listeners to handle user interactions. For example, selecting a node in the **JCheckBoxTree** triggers an event that updates the information displayed in the data panel. Buttons such as "Add Attribute" and "Display Attributes" trigger corresponding actions.
7. **Unique Attributes Manager:** The class interacts with the **UniqueAttributesManager** instance to manage and display unique attribute names.
8. **Groups Panel:** The class includes a **Groups** panel, which is a separate component for managing groups of nodes.

**Usage:**

Developers can use the **ProjectWindow** class to create a graphical user interface for managing a hierarchical tree structure with associated attributes. Key interactions and actions include:

* Adding new nodes to the tree.
* Deleting selected nodes from the tree.
* Grouping selected nodes.
* Adding attributes to selected nodes and managing their attribute tables.
* Displaying attributes associated with selected nodes.
* Displaying a list of unique attribute names.
* Selecting all nodes in the tree.
* Selecting nodes based on attribute conditions.

**Conclusion:**

The **ProjectWindow** class provides an intuitive and interactive user interface for managing a tree structure with associated attributes. By combining various panels, event listeners, and components, this class enables users to easily navigate the tree, manage attributes, and perform actions on nodes.

**7. Internal**: Main Class to Launch Internal Frames

The **Internal** class is a fundamental component of the multi-functional GUI application, facilitating the creation and display of internal frames within the graphical user interface. This report provides a comprehensive analysis of the **Internal** class, detailing its structure, functionalities, and role within the broader application context.

**Class Overview:**

The primary purpose of the **Internal** class is to manage the creation and display of internal frames, each hosting a specific graphical panel or component. This class serves as the entry point of the application and contributes to the modular structure by allowing multiple instances of the application's main functionality to run concurrently.

**Key Components:**

**Methods:**

1. **Main Method:**
   * The **main** method is the entry point of the application.
   * It utilizes the **SwingUtilities.invokeLater** method to ensure proper GUI initialization on the Event Dispatch Thread (EDT).
   * It invokes the **createAndShowGUI** method to set up and display the main graphical components.
2. **createAndShowGUI Method:**
   * This method sets up the main application window.
   * It creates a **JFrame** instance with a title, size, and layout.
   * It creates three internal frames using the **createInternalFrame** method.
   * A **JDesktopPane** is employed to contain the internal frames.
   * The desktop pane is added to the main frame, which is then set to be visible.
3. **createInternalFrame Method:**
   * This method generates an internal frame based on a specified title and color.
   * The frame is created with dimensions and a background color.
   * Depending on the title, a specific panel is added to the internal frame.
   * For instance, if the title is "Tree," a **ProjectWindow** panel is added.
   * The internal frame is made visible, and it is then returned.

**Functionality:**

1. **Internal Frame Creation:**
   * The application creates a main frame with a container for internal frames.
   * Using the **createInternalFrame** method, internal frames are generated with specific titles and colors.
2. **ProjectWindow Panel Integration:**
   * The **ProjectWindow** panel is added to the first internal frame.
   * The **ProjectWindow** presumably represents the main user interface for managing hierarchical tree structures.
3. **Modularity and Concurrent Functionality:**
   * The internal frames allow the application to compartmentalize functionalities.
   * Multiple instances of the application's core functionality can run concurrently within individual internal frames.

**Conclusion:**

In conclusion, the **Internal** class serves as the gateway to the multi-functional GUI application, enabling the creation and display of internal frames hosting various panels. By adopting an organized and modular approach, this class allows for the parallel execution of the application's main functionality across multiple internal frames. The class underscores the effective utilization of Swing components to manage and present distinct features of the application to users.

**8. Conclusion**

In conclusion, the multi-functional GUI application leverages Java Swing to provide users with an intuitive interface for managing hierarchical data structures, attributes, and groupings. The combination of custom components, table models, and managers facilitates efficient data organization and manipulation. The modular architecture and interactive design make the application a powerful tool for various data management tasks.

*Limitations:*

1. **Limited Attribute Management**: The project offers basic attribute management, but it lacks features like editing and updating attributes.
2. **Complex Data Handling**: The system deals only with simple attribute data types. Handling more complex data types like images or files would require additional development.
3. **User-Friendly Interface**: While the GUI is functional, there could be improvements in terms of user experience and visual design.

*Degree of Difficulty:*

Considering I have not worked with Java before and need to manage my schoolwork alongside the project, the degree of difficulty can be challenging. Learning a new programming language and developing a GUI application can be time-consuming. However, my progress in overcoming the issues and fixing the project demonstrates my ability to learn and problem-solve effectively. The project's complexity is moderate, given its focus on GUI development and data management. Balancing my schoolwork while working on the project shows dedication and multitasking skills.

**USE OF AND or OR:**

I’ve implemented a method **findNodesWithAttributes** that recursively searches for nodes within a tree structure that match specified attribute conditions. I am using a combination of attribute names and values to perform these searches. However, the code as it stands only supports using either the OR operator or the AND operator for combining attribute conditions, but not a combination of both.

The reason for this limitation lies in the structure of my conditional statements. Let's break down the key part of the code where the attributes are checked:

CODE:

if (node != null) {

AttributesTableModel attributesTableModel = nodeToAttributesTableModel.get(node);

System.out.println(node.getUserObject().toString());

if (attributesTableModel != null

&& (attributesTableModel.hasAttribute(attributeName1, attributeValue1)

|| attributesTableModel.hasAttribute(attributeName2, attributeValue2)

|| attributesTableModel.hasAttribute(attributeName3, attributeValue3))) {

resultNodes.add(node);

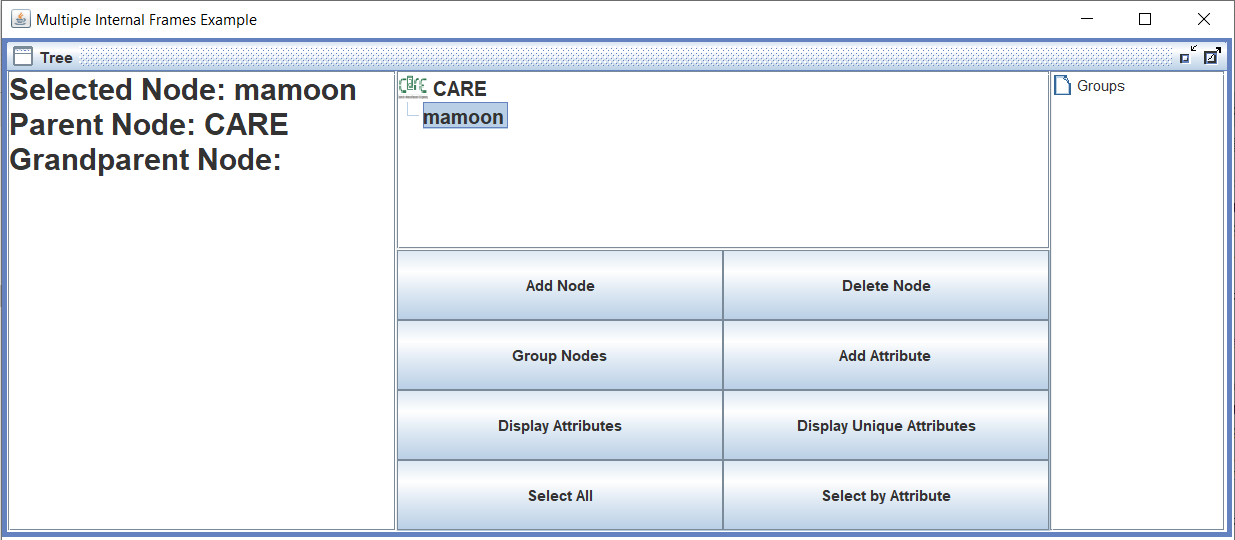
}

In this block of code, I am checking whether the attributesTableModel has any of the specified attribute conditions. If you think about how the logical operators work:

* Using **||** (OR): This will result in the node being added to the **resultNodes** list if it matches any of the specified attribute conditions. In this case, it's essentially an OR condition, and any match will trigger the addition of the node to the result.
* Using **&&** (AND): This will result in the node being added to the **resultNodes** list if it matches all of the specified attribute conditions. In this case, it's an AND condition, and all conditions must be met for the node to be included in the result.

Because I am using separate conditional checks for each attribute condition, there's no direct way to implement a combination of AND and OR operations for these conditions within this block of code. For instance, the code as it stands can't directly support something like "((A AND B) OR C)".

**Pictures:**



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A white screen with a white background

Description automatically generated with medium confidence

***-------------------------------END OF REPORT-------------------------------***