Document Tree 1.1 Component Specification

1. Design

The Document Tree component provides a SWING tree that allows the user to select model elements and groups of model elements. It also provides a way to signal the user of name changes and selections. It provides three kinds of views - diagram view, package view, and class view. It also allows for multiple tabs so that different kinds of views may be viewed on different tabs.

The primary interface of the design is the DocumentTreeTabbedPane interface which acts as a facade and hides the complexity of the component from the user. This interface represents a tabbed pane containing document tree panels. Each panel consists of a combo-box to choose between different types of views and a tree-view that shows the chosen view. It also has buttons to close that tab and create a new tab. The tree view itself is abstracted in a different interface, with different implementations showing different kinds of views. This view architecture also uses a few anonymous listener classes that allow listening to changing of node names and also selections.

There is no model explicitly implemented by this component. The default tree model provided by swing is used. We have our own tree node class which simply holds the actual element (graph element, model element or otherwise) that is in a tree node. We also define our own listener interface and events which can be used by applications to listen to selection and name change events.

This component also provides drag support, allowing tree nodes to be dragged out from the tree view. Note that JTree already has support to act as a drag source. The data flavor of our drag is simply the JVM local object corresponding to an array of DocumentTreeNode objects. This can be constructed as follows.

```
new DataFlavor(DataFlavor.javaJVMLocalObjectMimeType +

               ";class=\"" + DocumentTreeNode[].class.getName() + "\"");
```

We simply enable drag in the JTree and set a custom transfer handler.

In the version 1.1 overall layout will be updated to match updates to the overall application.

- In the diagram tree currently, there is a drop-down to choose new tabs to add to the tree view, as well as “Add” and “Close” buttons. The drop-down, “Add”, and “Close” buttons should be removed.
- The tabs shown in the view are configurable, so we are able to change the tabs and order of the tabs shown without having to change the code.
- In spite of all changes above, old version of component is fully functional and can be used too. Moreover, using the old version doesn’t bring any headaches to the user: he need to change nothing migrating from 1.0 to 1.1 (even configs may be left unchanged).

1.1 Design Patterns

Model-View-Controller Pattern - This pattern is used to separate the model from the view. The controller is handled implicitly by Swing.

Facade Pattern - The DocumentTreeTabbedPane interface acts as a facade and hides the complexity of the underlying component.

Observer Pattern - This pattern is used to observe the events happening in the GUI through the use of listeners.

Strategy Pattern – Used to plug new (1.1) implementation of DocumentTreePanel into whole component and leave old implementation still working.

1.2 Industry Standards

JFC / Swing.

1.3 Required Algorithms

1.3.1 Manipulating the tree

All the XXXDocumentTreeView classes need to manipulate tree structures so that a tree structure can be displayed. The developer should first acquaint himself with the javax.swing.tree package which contains the interfaces and classes we will be using for our tree structure. Note that the actual reference to tree nodes will be held by the BaseDocumentTreeView class.

All three tree manipulations are similar because, in essence we do traversal of some tree-like structure in the UML model and translate it into a tree for our purpose. The difference lies in which nodes are handled by which views. For example the Class View would not concern itself with diagram elements while the Diagram View wouldn't concern itself with model element hierarchy.

- Initialization - Initialization always consists of adding the element which acts as the root of the tree.
- Addition - Whenever a node is to be added to our tree, it implies that all nodes under the node need to be added and not just that node. Addition of a single node is done as follows.
  - Creating a DefaultMutableTreeNode which contains as a user object, the DocumentTreeNode to be added.
  - Add this tree node to the tree model under the suitable parent.
  - Add it to the two member maps storing DocumentTreeNode vs. DefaultMutableTreeNode and the reverse relationship.
  - Fire the appropriate event in the tree model.
- Removal - Similarly, whenever a node is removed, it implies that all its child nodes also need to be removed. Removal of a single node is as follows.
  - From the member map, get the DefaultMutableTreeNode to be removed. Remove it from the tree model.
  - Remove this node from the two member maps.
  - Fire the appropriate event in the tree model.
- Update - When a node is updated, there are two possibilities. If only its name has changed, we simply fire the appropriate event in the tree model. However, if its parent namespace has changed, we remove it and then add it back into the tree.

Now that we are comfortable with the basic process of tree node manipulation, we shall describe how the hierarchies are traversed for each tree view. The developer must have a certain degree of familiarity with the UML components before proceeding below. A moderate understanding of the Diagram Interchange, UML Model Core, UML Model Management, UML Model Manager, UML Model Activity Graphs, UML Model Collaborations is required.

- Diagram Centric - The root is the UML model, under it we have a layer of diagram categories which do not correspond to any entities but are simply strings used to group diagrams together. Under each diagram category, we have diagrams. The category of a diagram is given by its SimpleSemanticModelBridge.typeInfo member. Note that the Diagram class is also a GraphElement class and Graph Element contains Diagram Elements as children. This is the hierarchy which we are going to use for our tree. Therefore we start with the root Graph Element (the diagram in our case) and traverse all its children Diagram Elements. Any children which are also Graph Element objects are traversed further and so on.

  A GraphElement which has a UML1SemanticModelBridge is a possible candidate for entering the tree. It enters the tree if the underlying ModelElement is a package, class, interface or relationship. Note that all graph elements are added directly under the diagram and the depth of this tree is therefore 4 at most.

  For the add/update/remove tree node methods externally called, this view will ignore all document tree nodes that are not diagram categories or graph elements. A brief pseudo-code of the procedure to traverse the hierarchy is as follows.

```
void traverse(GraphElement element)
{
  if(element.semanticModelBridge instanceof Uml1SemanticModelBridge)
  {
    ModelElement me = (ModelElement) bridge.element;
    if (me instanceof {Package,Class,Interface,Relationship})
      add this graph element
  }
  List<DiagramElements> children = element.containeds;
  for each child
  {
    if(child instanceof GraphElement)
      traverse(child)
  }
}
```

- Class Centric - The root is the UML model. The hierarchy we are going to use is that of namespaces. Each namespace contains a list of owned elements and this translates into our parent-child relationship. If the owned ModelElement is also a namespace, we traverse its children too and so on. Since the model itself is also a namespace, we can conveniently start from there.

  We also need to add the standard class data. This can be done by getting the project configuration from the model manager and getting the list of standard namespaces from it. The same traversal procedure as before needs to be applied to each namespace. These namespaces will be added under the model.

  For the add/update/remove tree node methods externally called, this view will ignore all document tree nodes that are not model elements. A brief pseudo-code of the procedure to traverse the hierarchy is as follows.

```
void traverse(ModelElement me)
{
  if(me instanceOf
     {Namespace,Classifier,AssociationEnd,Method,Attribute} )
  {
    add this me;
  }
  if(me instanceOf Namespace)
  {
    for each owned element
    {
      traverse(owned element);
    }
  }
  if(me instanceOf Classifier)
  {
    for each association end
    {
      traverse(association end);
    }
    for each feature
    {
      traverse(feature);
    }
  }
}
```

- Package Centric - A package centric view is simply a class view with additional nodes, from outside the hierarchy. After constructing the class centric view, we get the diagrams of the model and add it under their namespaces. Then we get the activity graphs of the model and also add these under their namespaces.

  For the add/update/remove tree node methods externally called, this view will ignore all document tree nodes that are not model elements, except for those graph elements that are diagrams.

  This implementation not supposed to be used in new TC UML Tool versions, but it still can be turned on through configuration.

1.4 Component Class Overview

DocumentTreeTabbedPane - This interface gives the contract for a tabbed pane, where each pane cntains a document tree panel.

DocumentTreeTabbedPaneImpl - This class implements the previous interface.

DocumentTreePanel - This interface gives the contract for a panel, which contains a tree view, and a combo-box to select which tree view is shown.

DocumentTreePanelImpl - This class implements the previous interface.

DocumentTreeView - This interface gives the contract for a tree view, which shows the UML model as a JTree.

BaseDocumentTreeView - This abstract class implements the previous interface, defining the functionalities common to all views.

DiagramCentricDocumentTreeView - This concrete extension of BaseDocumentTreeView shows a diagram-centric tree view of the UML model.

PackageCentricDocumentTreeView - This concrete extension of BaseDocumentTreeView shows a package-centric tree view of the UML model.

ClassCentricDocumentTreeView - This concrete extension of BaseDocumentTreeView shows a class-centric tree view of the UML model.

DocumentTreeViewType - This enumeration represents the type of a document tree view.

DocumentTreeNode - This class represents a node in a document tree view.

DocumentTreeNodeType - This enumeration represents the type of a node in a document tree view.

DocumentTreeEventListener - This interface lays the contract for listening to events occuring in the document tree.

NameChangedEvent - This class represents a name change event.

SelectionChangedEvent - This class represents a selection change event.

DocumentTreeNodeTransferable - This class is a transferable implementation for document tree node arrays.

DocumentTreeViewTransferHandler - This class is a custom transfer handler that handles transfers for DocumentTreeView classes.

IconTreeCellRenderer – This class provides custom icons for each tree node.

ViewChangedEvent –This class represents a view changed event.

SimpleDocumentTreePanel – This is new implementation of DocumentTreePanel (from version 1.1), meeting new layout requirements.

1.5 Component Exception Definitions

DocumentTreeConfigurationException - This exception represents an error during configuration of a class of this component.

java.lang.IllegalArgumentException - This exception is thrown to indicate an illegal method argument.

1.6 Thread Safety

This component is not thread safe. Almost all classes have mutable data as tree structures may change and at a higher level, tabs or listeners may be added/removed. There is no requirement for this component to be thread safe. In a multi-threaded scenario once the GUI has been realized, it is essential to access this component from a single thread (the AWT event dispatch thread) since there are many SWING components used, which are also thread unsafe.

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.5 is required for compilation and testing.

2.2 TopCoder Software Components

- Base Exception 1.0 - This excetion is used as a base exception for the exception of our component.
- Configuration Manager 2.1.5 - This component is used to read configuration properties.
- Object Factory 2.0.1 - This component is used to create various objects - such as the XXXDocumentTreeView classes as well as the TreeCellRenderer implementation - and plug them in dynamically.
- Diagram Interchange 1.0 - This component is used to show a diagram view of the document tree.
- UML Model Core 1.0 - This component is used to represent the class and package views. It contains the core UML elements that will be shown.
- UML Model Manager 1.0 - This component is used to manage the entire model and access the different parts of the model.
- UML Project Configuration 1.0 - This component represents the project configuration fmo which we get our standard class data.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.gui.trees.document

com.topcoder.gui.trees.document.impl

3.2 Configuration Parameters

3.2.1 Common properties

config_manager_specification_factory_namespace - The namespace to create the ConfigManagerSpecificationFactory from. **Required.**

3.2.2 Configuring the DocumentTreeTabbedPanelImpl class

tree_cell_renderer - The key to create the class to render tree cells suing the Object Factory. **Required.**

3.2.2.a Configuring default tabs for the panel

default_tabs – Each value of this key represents one tab to be opened by default. To be exact, it’s string – fully qualified class name of the DocumentTreeView implementation to use in the tab. Order of values in the configuration is the order of tabs in the panel. It’s **optional**, but if missed the configuration for old (1.0) style panel must be presented (see 3.2.3).

3.2.3 Configuring the DocumentTreePanelImpl class

view_types - This parent property holds as child properties the list of view types, their associated classes and the text to show in the combo-box to choose views. It's **optional**, but must be presented if no configuration for the new (1.1) style provided (see 3.2.2.a).

view_types.<child-property-name> - The name of a child property gives the view type as the name of DocumentTreeViewType enumeration. Using the static members of the Enum class, we should be able to get back the appropriate enum instance from the given name. **Required.** Example: DIAGRAM_CENTRIC

view_types.<child-property>.view_class - This sub-property of a child property gives the fully qualified class name to be used to construct this view type using the reflection. **Required.**

view_types.<child-property>.view_text - This sub-property of a child property gives the text to be used in the combo-box to describe this view type. **Required.** Example: "Diagram-Centric" (quotes for clarity)

3.2.4 Configuring the IconTreeCellRenderer class

icons - This parent property holds as child properties the list of classes to add icons to, along with their associated icons. **Required.**

icons.<child-property-name> - The name of a child property gives the fully qualified class name of the class to which this icon must be selected. **Required.** Example: com.topcoder.uml.model.core.Namespace

icons.<child-property-value> - The name of a child property gives the path where the icon may be found. **Required.** Example: ../conf/icons/namespace.gif

3.2.5 Configuring the DiagramCentricDocumentTreeViewclass

diagram_categories - This parent property holds as child properties the list of diagram type info to diagram category name mappings. **Required**.
diagram_categories.<child-property-name> - The name of a child property gives the type info to be mapped. **Required**.
diagram_categories.<child-property-value> - The value of a child property gives the name of the diagram category for this type info. **Required**.

3.2.6 Configuring the PackageCentricDocumentTreeView

activity_graphs_category_name – The name of the activity graphs category, **Required**
collaborations_category_name – The name of the collaborations category, **Required**

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

See Demo.

4.3 Demo

4.4 Demo for a classic view

```
UMLModelManager model = TestHelper.createUMLModelManager();
```

```
    ModelElement modelElement = new GuardImpl();

    // Create a new tabbed pane object
    DocumentTreeTabbedPane treeTabbedPane = new DocumentTreeTabbedPaneImpl(model,
  TestHelper.NAMESPACE);

    // Get the tabbed pane and add it onto a container
    JPanel panel = new JPanel();
    panel.add(treeTabbedPane.getTabbedPane());

    // We may programmatically add/remove panes, get/set the chosen view as well as set the
  selected nodes
    // In the beginning, we start with three tabs
    // There is already a default tab, so we add two
treeTabbedPane.createNewPanel().setCurrentView(DocumentTreeViewType.PACKAGE_CENTRIC);
treeTabbedPane.createNewPanel().setCurrentView(DocumentTreeViewType.DIAGRAM_CENTRIC);
treeTabbedPane.createNewPanel().setCurrentView(DocumentTreeViewType.CLASS_CENTRIC);

    // When user clicks a model element in the diagram
    DocumentTreeNode node = new DocumentTreeNode(modelElement);
    treeTabbedPane.setSelectedTreeNodes(new DocumentTreeNode[] {node});

    // we may also add a listener to listen to events
    DocumentTreeEventListener myListener = new SimpleDocumentTreeEventListener();
  treeTabbedPane.addDocumentTreeEventListener(myListener);
```

4.5 Demo for a new (from version 1.1) view.

```
    UMLModelManager model = TestHelper.createUMLModelManager();

    // Create a new tabbed pane object
    DocumentTreeTabbedPane treeTabbedPane = new DocumentTreeTabbedPaneImpl(model,
TestHelper.NAMESPACE_NEW);

    // Get the tabbed pane and add it onto a container
    JPanel panel = new JPanel();
    panel.add(treeTabbedPane.getTabbedPane());

    // We may can't programmatically add/remove tabs, but can set
    // In the beginning (with sample configuration), we start with three tabs
    treeTabbedPane.getTabbedPane().setSelectedIndex(0);
    treeTabbedPane.getTabbedPane().setSelectedIndex(1);
    treeTabbedPane.getTabbedPane().setSelectedIndex(2);

    // select a tree node
    DefaultMutableTreeNode root = (DefaultMutableTreeNode)
        treeTabbedPane.getCurrentPanel().getCurrentView().getTree().getModel().getRoot();
    treeTabbedPane.setSelectedTreeNodes(new DocumentTreeNode[] {(DocumentTreeNode)
root.getUserObject()});

    // we may also add a listener to listen to events
    DocumentTreeEventListener myListener = new SimpleDocumentTreeEventListener();
    treeTabbedPane.addDocumentTreeEventListener(myListener);
```

● Swing Demonstration

```
/**
 * <p>
 * This is a demo frame for using the functionality provided by this document tree component.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 1.0
 */
public class FrameDemo extends JFrame {
    /**
     * <p>
     * The content panel of this frame.
     * </p>
     */
    private JPanel jContentPane;

    /**
     * <p>
     * The tree tabbed pane of this frame.
     * </p>
     */
```

```java
    private DocumentTreeTabbedPaneImpl treeTabbedPane;

    /**
     * <p>
     * This is the default constructor.
     * </p>
     *
     * @throws Exception if any exception occurs
     */
    public FrameDemo() throws Exception {
        initialize();
    }

    /**
     * <p>
     * This method initializes the widgets in this frame.
     * </p>
     *
     * @throws Exception if any exception occurs
     */
    public void initialize() throws Exception {
        this.setContentPane(getJContentPane());
        this.setTitle("Frame Demo");
    }

    /**
     * <p>
     * This method initializes jContentPane.
     * </p>
     *
     * @return the content panel for this frame.
     *
     * @throws Exception if any exception occurs
     */
    public JPanel getJContentPane() throws Exception {
        if (jContentPane == null) {
            jContentPane = new JPanel();
            jContentPane.setLayout(new BorderLayout());

            treeTabbedPane = new DocumentTreeTabbedPaneImpl(TestHelper.createUMLModelManager(), "some_namespace");

            DocumentTreePanel treePanel = treeTabbedPane.getCurrentPanel();
            for (DocumentTreeView view : treePanel.getAllViews()) {
                view.getTree().addMouseListener(new DragMouseAdapter());
            }
            treeTabbedPane.addDocumentTreeEventListener(new DocumentTreeEventListenerImpl());

            jContentPane.add(new JScrollPane(treeTabbedPane.getTabbedPane()), BorderLayout.CENTER);
        }

        return jContentPane;
    }

    /**
     * <p>
     * This class extends MouseAdapter class and is used for DnD demonstration.
     * </p>
     *
     * @author TCSDEVELOPER
     * @version 1.0
     */
    private class DragMouseAdapter extends MouseAdapter {
        /**
         * <p>
         * Handles the mouse pressed event.
         * </p>
         *
         * @param e the mouse event.
         */
        public void mousePressed(MouseEvent e) {
            JComponent c = (JComponent) e.getSource();
            TransferHandler handler = c.getTransferHandler();
            handler.exportAsDrag(c, e, TransferHandler.COPY);
        }
    }

    /**
     * <p>
     * This class implements DocumentTreeEventListener interface and is used for tree event handling demonstration.
     * </p>
     *
     * @author TCSDEVELOPER
     * @version 1.0
     */
    private class DocumentTreeEventListenerImpl implements DocumentTreeEventListener {
        /**
         * <p>
         * This method is called to indicate to the listener that the selection of nodes has changed.
         * </p>
         *
         * @param event The event denoting the selection.
         */
        public void treeNodeSelectionChanged(SelectionChangedEvent event) {
            System.out.println("Received SelectionChangedEvent : selected nodes are "
                + Arrays.asList(event.getSelectedTreeNodes()));
        }

        /**
         * <p>
         * This method is called to indicate to the listener that the name of a node has changed.
         * </p>
         *
         * @param event The event denoting the change of name.
         */
        public void treeNodeNameChanged(NameChangedEvent event) {
            System.out.print("Received NameChangedEvent : The tree node is [" + event.getTreeNode()
                + "], The new name is [" + event.getNewName() + "].");
        }
    }

    /**
     * <p>
     * The entry of the demo program.
     * </p>
     *
     * @param args the arguments
     *
     * @throws Exception to JUnit
     */
    public static void main(String[] args) throws Exception {
        TestHelper.loadConfigurations();

        FrameDemo frameDemo = new FrameDemo();

        frameDemo.addWindowListener(new WindowAdapter() {
            /**
             * <p>
             * This method handles the situation when the frame is closing.
             * </p>
             *
             * <p>
             * The namespaces are cleared in this method.
             * </p>
             *
             * @param event a window event instance
             */
            public void windowClosing(WindowEvent event) {
                super.windowClosed(event);

                try {
                    TestHelper.clearConfig();
                } catch (UnknownNamespaceException e) {
                    // ignore
                }
            }

        });

        frameDemo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frameDemo.setSize(new Dimension(500, 500));
        frameDemo.setVisible(true);
    }
      }
```

5. Future Enhancements

More tree views may be added.