Diagram UML Class Elements 1.0 Component Specification

1. Design

The Diagram UML Class Elements component provides the graphical diagram elements representing the model elements specific to a class diagram.

The elements provided are the package, the class, the interface, the exception and the enumeration. The last four elements are very similar so they represented almost in the same manner. The elements extend from the base element from the Diagram Elements component, providing the visual aspect of each element.

The rendering of the class elements is made by using Swing library.

1.1 Design considerations

The design of this component simply implements the concrete Nodes and the concrete NodeContainer for abstract ones defined in the Diagram Elements component. The main purpose of design – is to encapsulate rendering of the class elements. And as required – additional reactions for the mouse events were implemented.

PackageNode is a concrete NodeContainer, which takes information from Package class from UML Model and from the associated GraphNode. ClassNode, InterfaceNode, ExceptionNode and EnumerationNode are concrete Node classes, which represent Class, Interface, Class and Enumeration classes from UML Model respectively. They take information from associated GraphNode object too.

ClassNode, InterfaceNode, ExceptionNode and EnumerationNode are very similar, except the slight difference in compartments. So a common class named as BaseNode is defined. This class defines common properties like stroke color, fill color, and etc. BaseNode also contains name compartment, stereotype compartment, namespace compartment, attributes and operation compartments, but the location and value of these compartments are determined by concrete classes. Node classes can fire boundary changed event.

PackageNode is very like BaseNode, but since it should extend from NodeContainer, it can't inherit attributes defined in BaseNode class. Besides similar functionality as BaseNode, PackageNode also supports Drag-And-Drop, and it can fire edge adding or node adding events.

To allow popup menu, PopupMenuTrigger is defined. It is a mouse event listener, and it would show popup menu if some JComponent is right clicked. This listener is registered to all nodes automatically to support popup menu.

The PopupGroupFieldTrigger is used to show popup menu on attributes or operations compartments. Only one popup instance shared by the all attributes and operation. This approach will greatly save memory.

To allow double-click-editing, TextBoxTrigger is defined. It is a mouse event listener. It will ask the diagram viewer to show edit box when double click event occurs. This listener should be registered automatically to the element, which allows double-click-editing.

1.2 Design Patterns

Observer Pattern – System events are listened in this component, and also custom events are triggered for application to listen.

Template Method Pattern – the BaseNode class defines abstract getPreferredGraphNodeSize method and notifyGraphNodeChange method, which are differently implemented in concrete child classes.

1.3 Industry Standards

JFC Swing

UML 2.0 Diagram Interchange

1.4 Required Algorithms

1.4.1 How to Calculate the Connection Point to the PackageNode

The Package element consists of the two rectangles: one for name/namespace/stereotypes compartments and the second for main body. The all areas around the graphical representation of the PackageNode are counted on the next table:

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td colspan="2" rowspan="2">10</td><td>11</td></tr>
<tr><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

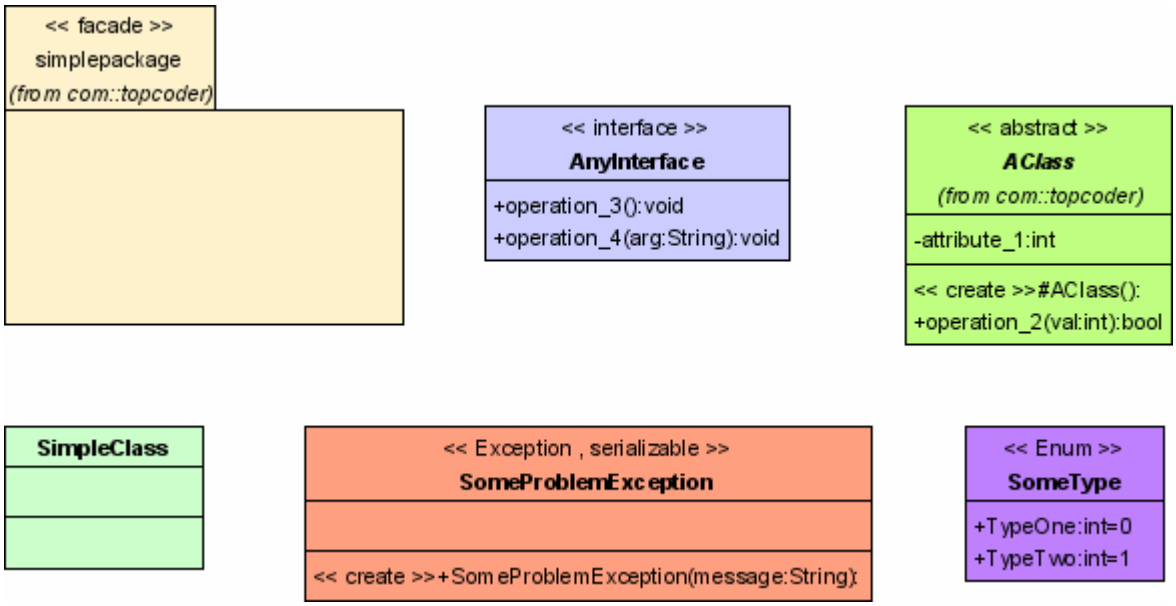
So the waypoint, which should be connected to the Package, can be in any of 15 areas. The table below describes to which rectangle we should connect the waypoint depending on its area.

Waypoint area	**Rectangle to connect**	
	Top (compartments)	**Bottom (main body)**
1	●	
2	●	
3		●
4		●
5	●	
6	●	
7		●
8		●
9		●
10		●
11		●
12		●
13		●
14		●
15		●

After the required rectangle to connect determined, the connection point on this rectangle side should be calculated by using interface from the next class (simply send the rectangle and the waypoint to it): com.topcoder.gui.diagramviewer.edges.RectConnector.

1.4.2 *How to Draw the Nodes*

A simple Poseidon-like drawing style implemented for the all nodes. Each node is based on several stacked rectangles. A quick reference picture is below.



1.4.3 *Non-trivial Rendering Issues*

When drawing elements of UML Class Diagram some issues should be considered. There are 3 slightly different algorithms to render all these elements:

1. Draw Package
2. Draw Interface, Class, Exception
3. Draw Enumeration

Package is rendered in two steps. First, we draw the left-top rectangle (where compartments are located). And next, the body rectangle should be painted. After retrieving position of the let-topmost corner and the bounds of the node (as JComponent properties) you should take in mind that the position relates to the compartments rectangle and the bounds relate to the entire size of the node. Therefore, to draw the first rectangle, the next actions should be implemented:

1. Use node position as the left-topmost corner of rectangle.
2. Get preferred size of the stereotype, name and namespace compartments.
3. Use only visible compartments for calculations.
4. Calculate width of rectangle as MAX(width of compartments).
5. Calculate height of rectangle as SUM(height of compartments + 20% gap between compartments).
6. Draw Rectangle2D of calculated bounds with the fillColor.
7. Draw Rectangle2D of calculated bounds with the strokeColor.
8. Note, the compartments will be painted automatically because they are child components.

To draw the second rectangle, the next actions should be implemented:

1. Use node position X coordinate for this rectangle position X.
2. Calculate position Y coordinate as node position Y + first rectangle height – 1.
3. Use node width for this rectangle width.
4. Calculate height as node height – first rectangle height + 1.
5. Draw Rectangle2D of calculated bounds and position with the fillColor.
6. Draw Rectangle2D of calculated bounds and position with the strokeColor.

Interface, Class, Exception are rendered by the same way in three steps. First, we draw the top rectangle (where name, namespace, and stereotype compartments are located). Second, the middle rectangle of attributes compartment should be painted. And next, the bottom rectangle of operations compartment will be shown. After retrieving position of the let-topmost corner and the bounds of the node (as JComponent properties) you should take in mind that the position relates to the first rectangle and the bounds relate to the entire size of the node. Therefore, to draw the first rectangle, the next actions should be implemented:

1. Use node position as the left-topmost corner of rectangle.
2. Get preferred size of the stereotype, name and namespace compartments.
3. Use only visible compartments for calculations.
4. Use node width for this rectangle width.
5. Calculate height of rectangle as SUM(height of compartments + 20% gap between compartments).
6. Draw Rectangle2D of calculated bounds with the fillColor.
7. Draw Rectangle2D of calculated bounds with the strokeColor.
8. Note, the compartments will be painted automatically because they are child components.

To draw the second rectangle, the next actions should be implemented (please note, this compartment has GroupTextField type and it is registered as a child graphical component, so it will be rendered automatically without additional calls from the paintComponent method of concrete node):

1. If the compartment is not visible draw nothing.
2. Get position and size of the compartment (as JComponent properties).
3. Draw Rectangle2D with the fillColor.
4. Draw Rectangle2D with the strokeColor.
5. Note, the all individual attributes will be painted automatically because they are child components.

To draw the third rectangle, the next actions should be implemented (please note, this compartment has GroupTextField type and it is registered as a child graphical component, so it will be rendered automatically without additional calls from the paintComponent method of concrete node):

1. If the compartment is not visible draw nothing.
2. Get position and size of the compartment (as JComponent properties).
3. Draw Rectangle2D with the fillColor.
4. Draw Rectangle2D with the strokeColor.
5. Note, the all individual operations will be painted automatically because they are child components.

Enumeration is rendered in four steps. First, we draw the top rectangle (where name, namespace, and stereotype compartments are located). Second, the middle top rectangle of enumeration literal compartment is drawn. Third, the middle bottom rectangle of attributes compartment should be painted. And next, the bottom rectangle of operations compartment will be shown. After retrieving position of the let-topmost corner and the bounds of the node (as JComponent properties) you should take in mind that the position relates to the first rectangle and the bounds relate to the entire size of the node. Therefore, to draw the first rectangle, the next actions should be implemented:

1. Use node position as the left-topmost corner of rectangle.
2. Get preferred size of the stereotype, name and namespace compartments.
3. Use only visible compartments for calculations.
4. Use node width for this rectangle width.
5. Calculate height of rectangle as SUM(height of compartments + 20% gap between compartments).
6. Draw Rectangle2D of calculated bounds with the fillColor.
7. Draw Rectangle2D of calculated bounds with the strokeColor.
8. Note, the compartments will be painted automatically because they are child components.

To draw the second rectangle, the next actions should be implemented (please note, this compartment has EnumerationLiteralCompartment type and it is registered as a child graphical component, so it will be rendered automatically without additional calls from the paintComponent method of concrete node):

1. If the compartment is not visible draw nothing.
2. Get position and size of the compartment (as JComponent properties).
3. Draw Rectangle2D with the fillColor.
4. Draw Rectangle2D with the strokeColor.
5. Note, the name and stereotype text fields will be painted automatically because they are child components.

To draw the third rectangle, the next actions should be implemented (please note, this compartment has GroupTextField type and it is registered as a child graphical component, so it will be rendered automatically without additional calls from the paintComponent method of concrete node):

6. If the compartment is not visible draw nothing.
7. Get position and size of the compartment (as JComponent properties).
8. Draw Rectangle2D with the fillColor.
9. Draw Rectangle2D with the strokeColor.
10. Note, the all individual attributes will be painted automatically because they are child components.

To draw the fourth rectangle, the next actions should be implemented (please note, this compartment has GroupTextField type and it is registered as a child graphical component, so it will be rendered automatically without additional calls from the paintComponent method of concrete node):

1. If the compartment is not visible draw nothing.
2. Get position and size of the compartment (as JComponent properties).
3. Draw Rectangle2D with the fillColor.
4. Draw Rectangle2D with the strokeColor.
5. Note, the all individual operations will be painted automatically because they are child components.

Please note, the locations of the all compartments should be processed in the notifyGraphNodeChange method of each concrete node.

1.4.4 How to Get Information for Nodes

This component is tightly coupled with two others: Diagram Elements and Diagram Interchange. These components provide all required information for each edge.

First component, Diagram Edges, represents a graphical side of the edge. Several base classes are inherited from the JComponent. The Node class defines common functionality for the all concrete nodes. For example, relative position, reference to the DiagramViewer, and mouse events processing. And the last important feature of the Node class is working with selection. The selection bound retrieved by the Diagram Elements component from the corresponding GraphNode UML model element automatically. The NodeContainer class implements actions to store several nodes.

The second component, Diagram Interchange, represents a UML model side of the node. Such data as position, size, and visibility can be directly retrieved from GraphElement, GraphNode, DiagramElement classes. The DiagramElement class contains a collection of properties, which accessed through the getProperties method. These properties relate to the font family, color and so on information. To be familiar with such data, the two another documents should be used. The common description for the Diagram Interchange Specification can be found in the "DiagramInterchangeSpec.pdf" file (or from the external link: http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-01.pdf). It clearly describes the purpose and features of the Diagram Interchange component. Another document is "**UML Tool - Class Diagram Elements Compartments.rtf**" file (or it can be found on the TopCoder forums). This file defines properties of the concrete UML Model elements, such as Class, Package and so on. First, the all elements of the UML diagram are linked in the graph. You can travel from leaves to the root and back on this hierarchy, because DiagramElement and GraphElement are linked together. GraphElement class has "containeds" list of child nodes, and each DiagramElement class has "container" reference to its parent. The related getters methods are implemented.

To get the type of the UML Model element, related to the GraphNode, you should get the "semanticModel" from the GraphElement, then convert its type to the Uml1SemanticModelBridge, and next retrieve the UML Model element by "getElement" method. Then you can convert the type of this element to the concrete type (and check for null), retrieve simple properties of the element (such as isAbstract and so on). Also consult "**UML Tool - Class Diagram Elements Compartments.rtf**" file for getting attributes and operations.

1.5 Component Class Overview

1.5.1 com.topcoder.gui.diagramviewer.uml.classelements namespace

PackageNode:

This class represents a Package element in UML Model. It allows to combine several nodes inside it. The presentaion and behaviour are similar to Java package of namespace in C#. It contains four properties. They are fill color, stroke color, font color, and font. Especially, JComponent#setFont and getFont methods are reused to support the font property.

Name compartment, stereotype compartment and namespace compartment are defined in this class. This class also registers EditBoxTrigger automatically to allow name editing. BoundChangeEvent would be triggered by this node. It indicates the node's bound is changed.

This class support Drag-And-Drop, and new node or edge adding event can be fired by this class.

PackageNodeConnector:

This class is the connector used to connect to PackageNode. It is relatively simple, as the shape of Package is two rectangles: one for name/namespace/stereotypes compartments and the second is for package body.

BaseNode:

This is the base Node of this component. It defines the common behaviors of all nodes in this component. The all nodes of this component, which are not containers, should inherit from this class. It contains four properties. They are fill color, stroke color, font color, and font. Especially, JComponent#setFont and getFont methods are reused to support the font property.

Name compartment, stereotype compartment and namespace compartment are defined in this class. This class also adds EditBoxTrigger in constructor automatically to allow name editing. BoundChangeEvent would be triggered by this node. It indicates the node's bound is changed.

ClassNode:

This class represents a Class node in UML diagram. It is an extension of BaseNode, which defines the most features for a node. This class only provides some specific methods to support the unique shape of class, to support the mouse events processing, and also to support the unique structure of class GraphNode.

ClassConnector:

This class is the connector used to connect to ClassNode. It is relatively simple, as the shape of class is a rectangle, some compartments are contained in the rectangle, and others are tightly stacked on the body rectangle.

InterfaceNode:

This class represents an Interface node in UML diagram. It is an extension of BaseNode, which defines the most features for a node. This class only provides some specific methods to support the unique shape of interface, to support the mouse events processing, and also to support the unique structure of interface GraphNode.

ExceptionNode:

This class represents a Class node (with default <<exception>> stereotype) in UML diagram. It is an extension of ClassNode, which defines the most features for a node. This class only provides some specific methods to support the mouse events processing.

EnumerationNode:

This class represents an Enumeration node in UML diagram. It is an extension of BaseNode, which defines the most features for a node. This class only provides some specific methods to support the unique shape of enumeration (with an additional EnumerationLiteral compartment), to support the mouse events processing, and also to support the unique structure of enumeration GraphNode.

EnumerationConnector:

This class is the connector used to connect to EnumerationNode. It is relatively simple, as the shape of class is a rectangle, some compartments are contained in the rectangle, and others are tightly stacked on the body rectangle.

EnumerationLiteralCompartment:

The special class representing combined text compartment for EnumerationNode. It contains two inner compartments: name and stereotype. They are created by the external class and will be registered as child components of the EnumerationLiteralCompartment class. So they will be shown automatically. This class only provides the bounding filled rectangle.

The coordinates of this class and its compartment are maintained by the external (contained) class.

GroupTextField:

The special class representing a group of text fields for XXXNode in this namespace. It contains a list of inner compartments: items. These text fields are created by the external

class and will be registered as child components of the GroupTextField class. So they will be shown automatically. This class only provides the bounding filled rectangle.

The coordinates of this class and its compartments are maintained by the external (contained) class.

StereotypeTextField:

The modified TextField class (with the stereotype attribute). The stereotype can be visible or hidden. This class will show the full text with stereotype. But on double click vent the only text field (without stereotype) will go to the edit control. So, the TextField.text will contain full string without stereotype. On the double click this string will go to the edit control. But in paintComponent function the concatenated string (stereotype + text) will be shown.

TextField:

Text field represents pure text compartment of Node or Edge. It could be used to represent name compartment, stereotype compartment and etc. Text field would be displayed as pure text with font and color inherited form parent node or edge. There is no decorator around or on the text.

It extends from com.topcoder.gui.diagramviewer.edges.TextField to allow to be added to Edge. After the Diagram Edges component is finally released, the methods and attributes already defined in base class should be removed in this class.

Please note, because this class contains all Font properties (through GraphNode reference). These properties include font size, font family name, and italics style (for abstract class name or methods).

1.5.2 com.topcoder.gui.diagramviewer.uml.classelements.event namespace

BoundaryChangedListener:

This interface defines the contract that every boundary change event listener must follow. It contains only one method to process the boundary changed event with a single BoundaryChangeEvent parameter.

BoundaryChangedEvent:

This is an event object used to indicate bound of node (including node container) is changed. It contains four properties. They are the Node whose bound is changed, the old bound value, the new bound value, and some message.

Note, the Node property can be retrieved by getSource().

EdgeAddListener:

This interface defines the contract that every edgeAdd event listener must follow. Note, the event would be triggered before the edge is actually added. This kind of listener can be registered to PackageNode instances. It contains only one method to process the edgeAdd event with a single EdgeAddEvent parameter.

EdgeAddEvent:

This is an event object used to indicate a new edge will be added to the PackageNode or BaseNode. It contains two properties. Location property tells where the edge ending should be added, and IsStart property tells whether the ending is a start or an end of edge. PackageNode or BaseNode can be retrieved by getSource(), and added edge type can be retrieved from diagram viewer.

NodeAddListener:

This interface defines the contract that every nodeAdd event listener must follow. Note, the event would be triggered before the node is actually added. This kind of listener can be registered to PackageNode instances. It contains only one method to process the node add event with a single NodeAddEvent parameter.

NodeAddEvent:

This is an event object used to indicate a new node will be added PackageNode or BaseNode. It contains only contains one location property, which tells where the new element should be added. PackageNode or BaseNode can be retrieved by getSource(), and added node type can be retrieved from diagram viewer.

TextChangeListener:

This interface defines the contract that every text change event listener must follow. Note, the event would be triggered before the text is actually changed. This kind of listener can be registered to TextField instances. It contains only one method to process the text change event with a single TextChangeEvent parameter. For example, application can register a listener to a TextField, which represents name compartment.

TextChangeEvent:

This is an event object used to indicate text of text field is changed. It contains three properties. They are the TextField whose text is changed, the old text value, the new text value. Note, the TextField property can be retrieved by getSource().

EditBoxTrigger:

This class can trigger edit box of diagram viewer when some component is double clicked. A TextField instance should be given to this class to tell which text field should be edited. For example, this trigger can be registered to a Class node, and the text field representing name compartment will be associated with this trigger. As a result, when the Class node is clicked, the name compartment will be editable.

EditBoxListener:

This listener is used to listen to events from edit box in diagram viewer. It must be attached to a TextField. It would fire a text change event when new text is entered, or display the original text if new text is canceled. This class is expected be used internally. It will be created in EditBoxTrigger#mouseClicked method, and will be registered to the edit box automatically.

PopupMenuTrigger:

This is an event listener which listens to mouse right clicked event. If the event occurs, popup menu would be shown. This event listener will be registered to every node or edge in this component automatically.

PopupGroupFieldTrigger:

This is an event listener which listens to mouse right button clicked event and the mouse left button clicked (selection). If the right button event occurs, popup menu would be shown. If the left mouse button clicked event occurs, then the corresponding text field will be selected. This event listener will be registered to every operations or attributes compartments in this component automatically.

1.6 Component Exception Definitions

The component defines a custom exception and reuses system-defined exceptions too.

IllegalGraphElementException:

This exception is used to indicate some GraphNode is illegal in specific situation. For example, if a Package GraphNode is given to the ClassNode constructor. It could be thrown when retrieving graph information from GraphNode.

Because this exception may be thrown in many places of application, we make it as a runtime exception. The other reason is that this exception can never happen in normal usage.

1.7 Thread Safety

This component is not thread-safe, because most of the classes in this component are mutable except the event classes. These event classes are thread safe because they are immutable. Thread-safety is not required. Like many other standard swing methods, thread-safety should be cared by users. And there is one issue we want to discuss further. Event listeners list is used by both the main

application thread (adding or removing listeners), and the event dispatching thread (using the listeners). This problem can be solved by listenerList field in JComponent, which is thread-safe. We add the listeners to the list, so thread safety is not a problem here.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.5
- Compile target: Java 1.5

2.2 TopCoder Software Components

- Diagram Elements 1.0 – the parent Node class (and basic node functionality) is used.
- Diagram Edges 1.0 – the basic functionality is used.
- Diagram Viewer 1.0 – allows editing name of nodes. Also is used for resizing of nodes. It holds the nodes defined in the class diagram.
- Diagram Interchange 1.0 – defines top-level parent classes (GraphNode, GraphEge and so on). It provides the standard UML diagram information, including size, position and etc.
- UML Model Core 1.0 – realization of DiagramElement. This component defines all the UML Model elements, like Class, Interface, and etc.
- UML Model Manager 1.0 – This component defines the Package interface.
- Base Exception 1.0 – contains the basic functionality for the custom exception.

Not used components from Requirement Specification – Configuration Manager 2.1.5

In this component, only GraphNode property name/key mapping requires configuration. Such configuration is provided as a java.util.Map instance to the constructor of concrete nodes. The functionality to load configuration from file should be done in the application level, because all kinds of nodes need such configuration in whole application.

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.diagramviewer.uml.classelements – contains all the class diagram elements.

com.topcoder.gui.diagramviewer.uml.classelements.event – contains all the listeners and events objects.

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

Put the dependent components under class path.

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

Add this package and the related components to the class path.

4.3 Demo

Note, this demo only shows part usage of this component, and the un-revealed part is much similar to following part.

```
// create custom properties mappings.
Map<String,String> properties = new HashMap<String,String>();
properties.put("StrokeColor", "stroke_color_property_key");
properties.put("FontSize", "font_size_property_key");
//...

// create a custom transfer handler class
public class CustomTransferHandler extends TransferHandler {
  // override 'importData' to provide custom transfering
  public boolean importData(JComponent comp, Transferable t) {
    // transfer 't' to this component,
    // maybe add a new element to diagram
  }
}

// create Package with properties mapping and transfer handler
PackageNode packNode = new PackageNode(<<graphNode>>, properties,
                                       new CustomTransferHandler());

// create a class to listen to node add event.
public class PackageChildAddListener implements NodeAddListener {
  public void nodeAdd(NodeAddEvent e) {
    // query the diagram viewer which kind of node should be added.
    Node element = << new element>>

    // specify the location of the element
    element.setLocation(e.getLocation());

    // get the package node instance from event
    PackageNode pkg = (PackageNode) e.getSource();

    // add the node to this package.
    pkg.addNode("BodyCompartment", <<new element>>);
  }
}

// register a listener to the package node to add child
packNode.addNodeAddListener(new PackageChildAddListener());

// create a class to listen to text change event.
public class NameChangeListener implements TextChangeListener {

  public void textChange(TextChangeEvent e) {
```

```
    // retrieve the TextField, and package node.
    TextField textField = e.getSource();
    PackageNode pkg = textField.getParent();

    // get package model element.
    GraphNode graphNode = pkg.getGraphNode();
    Package modelElement = graphNode.getModelElement();

    // actually change the name
    modelElement.setName(e.getNewText());

    // get preferred graphNode size
    Dimension newSize = pkg.getPreferredGraphNodeSize();

    // change size of graphNode according to new name
    graphNode.setSize(newSize);

    // notify the size of graphNode is changed.
    // the comment node will be updated accordingly.
    pkg.notifyGraphNodeChange();
  }
}

// register the text change listener to the name compartment.
// the name can be changed, and node will be resized automatically
packNode.getNameCompartment().addTextChangeListener(new NameChangeListener());

// Instantiate a concrete BaseNode
//(the example with ClassNode, all others - the same).
Map<String,String> propMap = new HashMap<String,String>();
propMap.put("FillColor", "fill_color_property_key");
propMap.put("FontColor", "font_color_property_key");
propMap.put("FontFamily", "font_family_property_key");
BaseNode bsNode = new ClassNode(some graph node, propMap);

// Retrieve compartments
TextField nameCompartment = bsNode.getNameCompartment();
TextField stereotypeCompartment = bsNode.getStereotypeCompartment();
TextField namespaceCompartment = bsNode.getNamespaceCompartment();
GroupTextField attributesCompartment = bsNode.getAttributesCompartment();
GroupTextField operationsCompartment = bsNode.getOperationscompartment();
```

5. Future Enhancements

- To implement another rendering scheme for nodes (for example, with shadows or 3D-like).