Diagram UML Sequence Elements 1.1 Component Specification

Additions will be in **RED** and updates in **BLUE**.

1. Design

The Diagram UML Sequence Elements component provides the graphical diagram nodes and edges representing the model elements specific to a sequence diagram. The rendering of the sequence elements is made by using Swing library.

Version 1.1adds the lifeline drawing functionality on the Object, an edge layout algorithm will be provided also. According to the current design, **each UML line was associated with only one edge**, thus the design keeps as previous on this issue.

The dragging functionality is already provided in Diagram Edge Component via the addEdgeDragListener method of Edge class.

1.1 Design considerations

The design of this component simply implements the concrete Node and concrete Edges for abstract ones defined in the Diagram Elements and the Diagram Edges components. The main purpose of design – is to encapsulate rendering of the sequence elements. And as required – an additional reaction for the mouse double-click event for the node was implemented.

The design is improving the requirements by defining and implementing two rendering schemes: SimpleScheme and TopCoderScheme. They are described in the Required Algorithms section. And some extended properties were defined.

In version 1.1, the newly added graph element classes keep the same style as 1.0. In addition to the requirements:

- New graph classes also support two rendering schemes
- Both Lifeline and LifelineSegment can be associated with a popup menu, the popup menu is configurable
- In addition to the required configurable properties, almost all properties of lifeline and lifeline segment can be configured via ConfigManager or at runtime.

1.2 Design Patterns

Template method pattern is used by the SequneceEdge providing a template for the concrete classes. And the same Template method pattern provided by ConfiguredEdgeEnding class for the concrete edge endings. The all Listeners and mechanism for their processing follow the Observer pattern.

Same as ObjectNode in 1.0, observer patterns are also used in Lifeline and LifelineSegment.

1.3 Industry Standards

JFC Swing

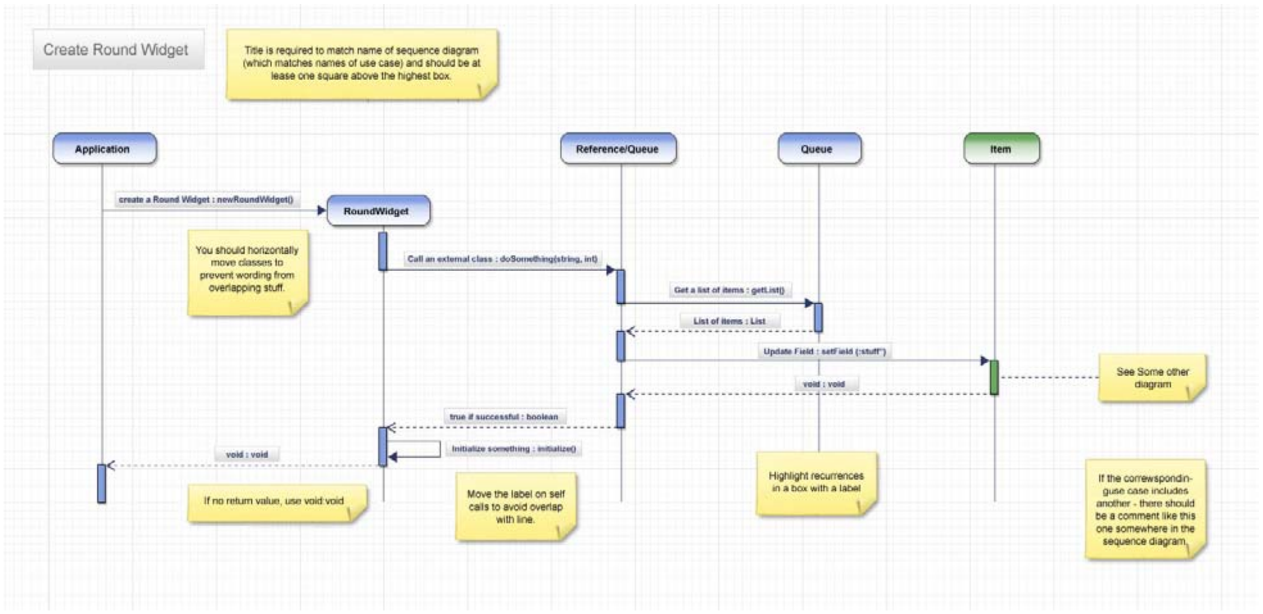
UML 2.0 Diagram Interchange

1.4 Required Algorithms

There are two types of used algorithms: one for graphical rendering and the second is for messages dispatching. Their descriptions are in the next chapters.

1.4.1 Graphical Rendering General Rules

The expected image of the Sequence Diagram is shown on the next picture. This image is also stored in the full scale in the diagram-sequence.jpg file.


Create Round Widget
Title is required to match name of sequence diagram (which matches names of use case) and should be at lease one square above the highest box.
Application
Reference/Queue
Queue
Item
create a Round Widget : newRoundWidget()
RoundWidget
You should horizontally move classes to prevent wording from overlapping stuff.
Call an external class : doSomething(string, int)
Get a list of items : getList()
List of items : List
Update Field : setField (:stuff")
See Some other diagram
void : void
true if successful : boolean
Initialize something : initialize()
void : void
If no return value, use void:void
Move the label on self calls to avoid overlap with line.
Highlight recurrences in a box with a label
If the correwspondin-guse case includes another - there should be a comment like this one somewhere in the sequence diagram.

The responsibility of the component is to show only **one node type and several edges**. But the TopCoder requirements for displaying sequence elements are complex – gradient fills, shadows and so on. This design introduces two rendering schemes to improve speed of the component coding/testing, to get the visual results faster, and for future enhancements. They are:

- SimpleScheme,
- TopCoderScheme.

The TopCoderScheme is shown on the previous picture. The Simple scheme removes all gradients, shadowing, round corners, textbox backgrounds and borders. This scheme can be very easily implemented and tested. And the benchmarking with the simple rendering scheme can give important information about the top limit speed. Although the properties and configuration parameters for both schemes are implemented, the component allows only implementing the simple scheme for the version 1.0 of the component at the development side, and the improved schemes as version 1.1 and so on.

1.4.2 Drawing the node

The algorithm for simple rendering scheme is:

- Draw the filled rectangle with the size of node by using fill color and stroke color.
- Draw the name of node in the center of the prepared rectangle by using the font color.

The algorithm for TopCoder scheme is more complex:

- Draw the filled rounded rectangle with the size of node by using shadow color. The top-left coordinates are adjusted by adding shadow length to both of them. The rectangle rounding should use rounding radius value.
- Draw the filled by gradient rounded rectangle with the size of node by using fill color, fill color2 and stroke color. The rectangle rounding should use rounding radius value.
- Draw the name of node in the center of the prepared rectangle by using the font color.

1.4.3 Drawing the lifeline

The algorithm for simple rendering scheme is:

- Draw the dashed line from the leftEnding endpoint to the rightEnding endpoint by using the stroke color.

The algorithm for TopCoder scheme is more complex:

- Draw the dashed line from the leftEnding endpoint to the rightEnding endpoint by using the stroke color.
- Draw the shadow of the line by using shadow color and shadow length.

1.4.4 Drawing the lifeline segment

The algorithm for simple rendering scheme is:

- Draw the filled rectangle with the size of lifeline segment by using fill color and stroke color.

The algorithm for TopCoder scheme is more complex:

- Draw the shadow of the rectangle by using shadow color and shadow length
- Draw the filled rectangle with the size of lifeline segment by using fill color and stroke color.

1.4.5 Drawing the edge

The algorithm for simple rendering scheme is:

- Draw the dashed line from the leftEnding endpoint to the rightEnding endpoint by using the stroke color. The line dashing spaces are configured by the lineStyle. If the blank length = 0, then the line is not dashed.
- Draw the name of edge in the center and upward of the drawn line by using font color.

The algorithm for TopCoder scheme is more complex:

- Draw the dashed line from the leftEnding endpoint to the rightEnding endpoint by using the stroke color. The line dashing spaces are configured by the lineStyle. If the blank length = 0, then the line is not dashed.
- Draw the filled with gradient rectangle (text box) in the center and upward of the drawn line by using the textbox stroke color, textbox fill color and textbox fill color2.
- Draw the name of edge in the center of the drawn textbox by using font color.

1.4.6 Drawing the edge ending

The algorithm for both simple and TopCoder rendering schemes is the same:

- Draw the arrow by using the endpoint as the origin, the stroke color for arrow lines, and the filling color for filling.

1.4.7 Layout edges (DefaultEdgeLayoutStrategy)

In the default implementation, the edges will be arranged like a chain. There will not be two edges drawn on the same level. The spacing between the edges will be configurable and it will be constant. Note that all the arrows start a new lifeline segment in the destination object. The **asynchronous message** is the only one that does not break the lifeline segment in the source lifeline. For all the other messages, the lifeline segments end in the position where the arrow starts.
Also note that the lifeline with the first edge will have a lifeline segments drawn from where the object header down to where the first arrow starts.

LS stands for lifeline segment
curObject means the ObjectNode from which the current edge starts
nextObject means the ObjectNode at which the current edge ends

The following pseudocode demonstrates the algorithm:

```
lsList = create a new list
curLS = null       // current processing LS
curY = 0           // current Y position, assume that the top is 0
for each edge in the specified edge list
```

```
            if this is the first edge
            then
                  // create the first special LS, the properties of this
                  // LS is from the corresponding ObjectNode
                  firstLS = create a LS with position.y=curObject.bottom
                                             position.x=curObject.center.x
                                             height=edgeSpace
                  append firstLS to lsList
                  // create current LS, the properties of this LS is from
                  // the link that starts this LS
                  curY = bottom of firstLS
                  curLS = create a LS with position.y=curY
                                              position.x=nextObject.center.x
                  curY += edgeSpace
                  // set the edge
                  edge.leftEnding.x = firstLS.x
                  edge.rightEnding.x = curLS.x
                  edge.y = curLS.y
                  // let's continue
                  continue loop
            endif
            // if we reach here, this is not the first edge
            if this is an asynchronous message
            then
                  // no lifeline segment break, just set the edge
                  edge.leftEnding.x = curLS.x
                  edge.rightEnding.x = nextObject.center.x
                  edge.y = curY
                  // add space
                  curY += edgeSpace
            else
                  // end current LS
                  curLS.height = curY – curLS.y
                  append curLS to lsList
                  // set the edge
                  edge.leftEnding.x = curLS.x
                  edge.rightEnding.x = nextObject.center.x
                  edge.y = curY
                  // create new LS
                  curLS = create a LS with position.y=curY
                                             position.x=nextObject.center.x
                  // add space
                  curY += edgeSpace
            end if
      end for
      // now we must end the final lifeline segment
      curLS.height = curY – curls.y
      append curLS to lsList
```

1.4.8 Messages Dispatching

The component dispatches user interface events for node and for the edges. The algorithms are different and they are described below.

The events processing for nodes are here:

- Single click on the node will select it. And the selection corners will be shown.
- If the visibility of the node name or **stereotypes compartment** were changed then the node need to be resized. In this case the special event to the external component (DrawingViewer) is sending.

- If the name of node was made empty – then the default value ("anonymous") should be shown.
- The node will react to a mouse double-click event, by showing the edit control of the DiagramViewer in order to edit the name of the node. The text of the Name compartment will not be shown while the edit control is up (though the node will not be resized). The event provides to the DiagramViewer the position where to show the edit control, so it fits on top of the Name compartment, and the initial text. It also registers a listener to receive the event that the text was entered or cancelled in the edit control. It will remove the listener after receiving the event.
- The new name will not be set after the end of name editing process. An event is generated instead, with the old and new name, and with the node and graph node for which the name is set. The node also provides a method to check the size that will be required for the node if the name would be set. The application will register for the event and, eventually it will set the new name. The method for setting the name performs the resize of the name, and it will generate a resize event (the reason is also passed, as a string).
- The node is not implemented as a drag and drop **DropTarget**, as it is not a container. However, it does not interfere with the drag and drop action initiated by the user. The user is able to drop the element on top of the node and the event is handled by the diagram behind it (as the intention of the user is to add the element to the diagram).
- In case the diagram viewer's flag for adding new elements from the toolbar is on, the node reacts to mouse events differently, by letting the events pass to the element behind.
- Right click on the edge will show the popup window (if it was registered).

The events processing for the edges are:

- Single click on the edge will select it. And the selection corners will be shown.
- Selecting the text fields will select the all edge.
- Double clicking on the edge does nothing.
- In case the diagram viewer's flag for adding new elements from the toolbar is on, the node reacts to mouse events differently, by letting the events pass to the element behind.
- Right click on the edge will show the popup window (if it was registered).

The events processing for the lifelines are:

- Right click on the edge will show the popup window (if it was registered).
- Single clicking on the lifeline segment does nothing.
- Double clicking on the lifeline segment does nothing.

The events processing for the lifeline segment are:

- Single click on the lifeline segment will select it; the border of the lifeline segment will be wider (specifically, +1 pixel).
- Double clicking on the lifeline segment does nothing.
- Right click on the lifeline segment will show the popup window (if it was registered).

1.5 Component Class Overview

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

ObjectNode:

The concrete implementation of the Node abstract class. This class is one and only node for sequence diagrams. The main purpose - to draw graphical representation of the node. The corresponding model element is stored inside the class. **Connecting edges accepted by the graphConnector of this class**. To configure the defaults - use ConfigManager configuration file. This file is processed by the static initialization block of the class.

Many events are processed by the class and some are generated. Much functionality of event processing logic is reused from the parent class.

EditControlHandler:

The concrete implementation of the EditControlListener interface. This class is registered to listen to the "edit finished" action on the name of the node. The top level component (DiagramViewer) implements this action. The current class will implement required actions on the end of name editing.

ResizeListener:

The interface for listening to the component resizes requests. The external component can implement this interface to be registered by the ObjectNode. This is a child of the EventListener - so it works in the standard way. The resizeNeeded message will occur when the ObjectNode need to be resized and therefore sending the message to the top level component.

ResizeEvent:

The event for resizing message. It includes an instance of the ObjectNode. Therefore the top-level component can easily perform needed actions on it.

SetNameListener:

The interface for listening to the component name change requests. The external component can implement this interface to be registered by the ObjectNode. This is a child of the EventListener - so it works in the standard way. The namePrepared message will occur when the ObjectNode need to change its name and therefore sending the message to the top level component.

SetNameEvent:

The event for name setting message. It includes an instance of the ObjectNode. Therefore the top-level component can easily perform needed actions on it.

EditNameListener:

The interface for listening to the component start name edits requests. The external component can implement this interface to be registered by the ObjectNode.This is a child of the EventListener - so it works in the standard way. The nameEditStarted message will occur when the ObjectNode need to start name editing action and therefore sending the message to the top level component.

EditNameEvent:

The event for starting name edits action. The position and current name of the ObjectNode are enough for external component to implement editing.

RenderScheme:

The enum representing several rendering algorithms to be used for sequence diagram elements drawing. It has two kinds of render schemes: SimpleScheme and TopCoderScheme.

The RenderScheme is also used in Lifeline and LifelineSegment.

SequenceEdge:

This abstract class extends the parent Edge class by configuration options. And some common behavior (popup registering, mouse events processing) for the all sequence diagram edges are implemented. To configure the defaults - use ConfigManager configuration file. This file is processed by the static initialization block of the class.

The class processes mouse events, but does not react to most of them. They are passed to the top-level classes. Only popup window showing is processed.

CreateMessageEdge:

The concrete edge for Create Message used on sequence diagrams. The painting of the edge is implemented.

SynchronousMessageEdge:

The concrete edge for Synchronous Message used on sequence diagrams. The painting of the edge is implemented.

AsynchronousMessageEdge:

The concrete edge for Asynchronous Message used on sequence diagrams. The painting of the edge is implemented.

SendSignalMessageEdge:

The concrete edge for Send Signal Message used on sequence diagrams. The painting of the edge is implemented.

ReturnMessageEdge:

The concrete edge for Return Message used on sequence diagrams. The painting of the edge is implemented.

ObjectNodePropertyType:

The enum representing all possible kinds of the used properties for the ObjectNode. The enum elements corresponds to the ObjectNode class instance variables with the same names: strokeColor, fillColor and so on.

SequenceEdgePropertyType:

The enum representing all possible kinds of the used properties for the SequenceEdge. The enum elements corresponds to the SequenceEdge class instance variables with the same names: strokeColor, fillColor and so on.

EdgeLayoutStrategy:

The EdgeLayoutStrategy interface encapsulates layout algorithm. This interface defines a single method layout() to layout edges.

SequenceEdgePropertyType:

The enum representing all possible kinds of the used properties for the SequenceEdge. The enum elements corresponds to the SequenceEdge class instance variables with the same names: strokeColor, fillColor and so on.

1.5.2 com.topcoder.gui.diagramviewer.uml.sequenceelements.lifeline namespace

LifelinePropertyType:

The enum representing all possible kinds of the used properties for the Lifeline. The enum elements corresponds to the Lifeline class instance variables with the same names: strokeColor, shadowColor and so on.

Lifeline:

The Lifeline class represent the lifeline of an object in sequence diagram. This class extends the parent Edge class by configuration options. To configure the defaults - use ConfigManager configuration file. This file is processed by the static initialization block of the class.

The class processes mouse events, but does not react to most of them. They are passed to the top-level classes. Only popup window showing is processed.

LifelineSegmentPropertyType:

The enum representing all possible kinds of the used properties for the LifelineSegment. The enum elements corresponds to the LifelineSegment class instance variables with the same names: strokeColor, fillColor and so on.

LifelineSegment:

The LifelineSegment class represents the lifeline segment in sequence diagram. This class is a concrete implementation of the Node abstract class. The main purpose - to draw graphical representation of the lifeline segment. To configure the defaults - use ConfigManager configuration file. This file is processed by the static initialization block of the class.

Many events are processed by the class and some are generated. Much functionality of event processing logic is reused from the parent class.

1.5.3 com.topcoder.gui.diagramviewer.uml.sequenceelements.edgelayout namespace

DefaultEdgeLayoutStrategy:

The default implementation of EdgeLayoutStrategy. This implementation layouts edges as a chain, please refer to the algorithm section for the details on this algorithm.

1.5.4 com.topcoder.gui.diagramviewer.uml.sequenceelements.edgeendings namespace

ConfiguredEdgeEnding:

This abstract class extends the parent EdgeEnding class by configuration options. To configure the defaults - use ConfigManager configuration file. This file is processed by the static initialization block of the class.

FilledArrowEdgeEnding:

The concrete edge ending with filled arrow used on sequence diagrams. The painting of the edge ending is implemented.

EmptyArrowEdgeEnding:

The concrete edge ending with empty arrow used on sequence diagrams. The painting of the edge ending is implemented.

NothingEdgeEnding:

The concrete edge ending with no arrow used on sequence diagrams. The painting of the edge ending is implemented.

HalfEmptyArrowEdgeEnding:

The concrete edge ending with half empty arrow used on sequence diagrams. The painting of the edge ending is implemented.

ConfiguredEdgeEndingPropertyType:

The enum representing all possible kinds of the used properties for the ConfiguredEdgeEnding. The enum elements corresponds to the ConfiguredEdgeEnding class instance variables with the same names: strokeColor, fillColor and so on.

1.6 Component Exception Definitions

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

IllegalArgumentException:

If the argument is null or empty then this exception will occur. In some cases empty values are allowed – refer to the class diagram descriptions for the details.

SequenceElementsConfigurationException:

If the default values loading functions will get a problem, then this exception will throw. **The inner exceptions from ConfigManager are chained by this exception too.**

EdgeLayoutException:

Thrown by EdgeLayoutStrategy to indicate that error occurs during layout.

1.7 Thread Safety

This component is not thread-safe, because most of the classes in this component are mutable except the event classes. 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 safe is not a problem here.

The newly added graph elements are not thread-safe the same reason as those in 1.0, the solution for making them thread-safe is the same as described above. The DefaultEdgeLayoutStrategy is not thread-safe since it's mutable.

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 parent Edge class (and basic node functionality) is used.
- Diagram Viewer 1.0 – allows to edit name of node and edges. Also is used for resizing of node and edges.
- Diagram Interchange 1.0 – defines top-level pranet classes (GraphNode, GraphEge, Rectangle, Point and so on).
- UML Model Core 1.0 – realization of DiagramElement.
- Configuration Manager 2.1.5 – provide the setup of default values for node, edges, and edge endings properties.
- Base Exception 1.0 – allows to inherit custom exceptions in the useful way.

Several components are suggested by the Requirement Specification: Action Manager 1.0, UML Model Manager 1.0, UML Project Configuration 1.0. But were unused. They are helpful for understanding of internal mechanisms of TopCoder UML Tool, but useless for this component design.

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.sequenceelements – main classes and interfaces, including the edge layout interface com.topcoder.gui.diagramviewer.uml.sequenceelements.edgeendings – the concrete edge endings classes.

com.topcoder.gui.diagramviewer.uml.sequenceelements.lifeline – lifeline classes

com.topcoder.gui.diagramviewer.uml.sequenceelements.edgelayout – concrete edge layout algorithms

3.2 Configuration Parameters

The default properties of the node, edges, edge endings, lifeline, and lifeline segment are configured by using Configuration Manager. The list of properties for a node is below. The default values are from Requirement Specification, and they can be easily changed for more realistic (especially colors) in future. The font size is in points. The all lengths of graphical objects are in pixels.

Parameter	Description	Values
NodeStrokeColor	The color used for drawing foreground items (except text)	Color.BLACK Optional
NodeFillColor	The color used for background	Color.BLACK Optional
NodeFillColor2	The color used for gradient fills of background	Color.BLACK Optional
NodeShadowColor	The color used for shadow drawing	Color.BLACK Optional
NodeFontColor	The color for text drawing	Color.BLACK Optional
NodeFontFamily	The family of font used for text drawing	“Arial” Optional
NodeFontStyle	The style of font used for text drawing	Font.PLAIN Optional
NodeFontSize	The point size of the font used for text drawing	10 Optional
NodeRenderScheme	The style of node rendering (possible value: 0 – SimpleScheme, 1 – TopCoderScheme)	1 Optional
NodeWidth	The width of the node	100 Optional
NodeHeight	The height of the node	40 Optional
NodeWidthMinimum	The minimum limit of the node width	60 Optional
NodeHeightMinimum	The minimum limit of the node height	40 Optional
NodeRoundingRadius	The radius of the node rounding	8 Optional
NodeShadowLength	The length of shadow	5 Optional

The list of properties for an edge is below. The default values were made the same as node defaults from Requirement Specification, and they can be easily changed for more realistic (especially colors) in future. The font size is in points. The all lengths of graphical objects are in pixels.

Parameter	Description	Values
EdgeStrokeColor	The color used for drawing foreground items (except text)	Color.BLACK Optional
EdgeFillColor	The color used for background (except textbox item)	Color.BLACK Optional
EdgeFillColor2	The color used for gradient fills of background (except textbox item)	Color.BLACK Optional
EdgeShadowColor	The color used for shadow drawing	Color.BLACK Optional
EdgeFontColor	The color for text drawing	Color.BLACK Optional
EdgeTextBoxStrokeColor	The color for textbox foreground items (except text)	Color.BLACK Optional
EdgeTextBoxFillColor	The color used for text background	Color.BLACK Optional
EdgeTextBoxFillColor2	The color used for gradient fills of text background	Color.BLACK Optional
EdgeFontFamily	The family of font used for text drawing	“Arial” Optional
EdgeFontStyle	The style of font used for text drawing	Font.PLAIN Optional
EdgeFontSize	The point size of the font used for text drawing	10 Optional
EdgeRenderScheme	The style of edge rendering (possible value: 0 – SimpleScheme, 1 – TopCoderScheme)	1 Optional
EdgeDashLength	The length of filled part of a dash line	4 Optional
EdgeBlankLength	The length of blanck part of a dash line	4 Optional

The list of properties for a lifeline segment is below. The all lengths of graphical objects are in pixels. Note that the format for color must be “#RRGGBB”.

Parameter	Description	Values
LifelineSegmentStrokeColor	The color used for drawing border	#000000 Optional
LifelineSegmentFillColor	The color used for filling inside	#000000 Optional
LifelineSegmentShadowColor	The color used for shadow drawing	#000000 Optional
LifelineSegmentRenderScheme	The style of lifeline segment rendering (possible value: 0 – SimpleScheme, 1 – TopCoderScheme)	1 Optional
LifelineSegmentWidth	The width of the lifeline segment	5 Optional
LifelineSegmentHeight	The height of the lifeline segment	15 Optional
LifelineSegmentWidthMinimum	The minimum limit of the lifeline segment width	3 Optional
LifelineSegmentHeightMinimum	The minimum limit of the lifeline segment height	5 Optional
LifelineSegmentShadowLength	The length of shadow	5 Optional

The list of properties for a lifeline is below. The all lengths of graphical objects are in pixels. Note that the format for color must be “#RRGGBB”.

Parameter	Description	Values
LifelineLength	The length of the lifeline	100 Optional
LifelineStrokeColor	The color used for drawing the lifeline	#000000 Optional
LifelineShadowColor	The color used for shadow drawing	#000000 Optional
LifelineRenderScheme	The style of lifeline rendering (possible value: 0 – SimpleScheme, 1 – TopCoderScheme)	1 Optional
LifelineDashLength	The length of filled part of a dash line	4 Optional
LifelineBlankLength	The length of blank part of a dash line	4 Optional
LifelineShadowLength	The length of the shadow length	5 Optinal

The list of properties for DefaultEdgeLayoutStrategy is below. The length is in pixels.

Parameter	Description	Values
LayoutEdgeSpace	The space between edges	20 Optional

The list of properties for an edge ending is below. The all lengths of graphical objects are in pixels.

Parameter	Description	Values
EdgeEndingStrokeColor	The color used for drawing	Color.BLACK Optional
EdgeEndingFillColor	The color used for background	Color.BLACK Optional
EdgeEndingRenderScheme	The style of edge rendering (possible value: 0 – SimpleScheme, 1 – TopCoderScheme)	1 Optional
EdgeEndingArrowXLength	The X length (leg) of the ending arrow	10 Optional
EdgeEndingArrowYLength	The Y length (leg) of the ending arrow	7 Optional

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

The most expected scenario of using this component is described by the source code below.

```
// Create the node
ObjectNode objNode = TestHelper.createObjectNode();

// Work with properties
objNode.setName("NewClass");
objNode.setIsStereotypesVisible(false);
objNode.setIsNameVisible(true);
objNode.setFontFamily("Times New Roman");
objNode.setFontSize(objNode.getFontSize() + 2);
objNode.setFontColor(Color.RED);
objNode.setFontStyle(Font.BOLD);
objNode.setStrokeColor(Color.GREEN);

// Work with size and position
Dimension size = objNode.getPrefferedSize("NewInterface", true, true);
objNode.setSize(size);
objNode.setName("NewInterface");
objNode.setIsStereotypesVisible(true);
objNode.setPosition(new Point(50, 20));

// Add popup menu
JPopupMenu popup = new JPopupMenu();
objNode.setPopup(popup);

// Register to the listeners
ResizeListener rszHandler = new MockResizeListener();
objNode.addResizeListener(rszHandler);
EditNameListener editNameHandler = new MockEditNameListener();
objNode.addEditNameListener(editNameHandler);
SetNameListener setNameHandler = new MockSetNameListener();
objNode.addSetNameListener(setNameHandler);

// Prepare the other objects for edge instantiating
Link link = new LinkImpl();

// set up the GraphEdge instance
Diagram diagram = new Diagram();
GraphEdge graphEdge = new GraphEdge();
graphEdge.addWaypoint(TestHelper.createDiagramInterchangePoint(100, 100));
graphEdge.addWaypoint(TestHelper.createDiagramInterchangePoint(200, 200));
graphEdge.addWaypoint(TestHelper.createDiagramInterchangePoint(300, 400));
diagram.addContained(graphEdge);

// Instantiate different edges with different edge endings
CreateMessageEdge createMessageEdge = new CreateMessageEdge(link, graphEdge,
    TestHelper.createEdgeEndingProperties(), TestHelper.createSequenceEdgeProperties());
SynchronousMessageEdge synchMessageEdge = new SynchronousMessageEdge(link, graphEdge,
    TestHelper.createEdgeEndingProperties(), TestHelper.createSequenceEdgeProperties());
AsynchronousMessageEdge asynchMessageEdge = new AsynchronousMessageEdge(link, graphEdge,
    TestHelper.createEdgeEndingProperties(), TestHelper.createSequenceEdgeProperties());
SendSignalMessageEdge sendSigMessageEdge = new SendSignalMessageEdge(link, graphEdge,
    TestHelper.createEdgeEndingProperties(), TestHelper.createSequenceEdgeProperties());
ReturnMessageEdge retMessageEdge = new ReturnMessageEdge(link, graphEdge,
    TestHelper.createEdgeEndingProperties(), TestHelper.createSequenceEdgeProperties());

// Add popup menu
JPopupMenu popup1 = new JPopupMenu();
createMessageEdge.setPopup(popup1);
synchMessageEdge.setPopup(popup1);
asynchMessageEdge.setPopup(popup1);
sendSigMessageEdge.setPopup(popup1);
 retMessageEdge.setPopup(popup1);
```

```
/*

 * Test for Lifeline,LifelineSegment and EdgeLayoutStrategy
 */

 //Lifeline and LifelineSegment are generated by the component, end user needn't
  //care about how to create them.

LifelineSegment segment = TestUtil.createLifelineSegment();
Lifeline line = TestUtil.createLifeline();

//Set properties of a lifeline segment and a lifeline
 segment.setShadowLength(3);
 segment.setStrokeColor(Color.YELLOW);
 line.setStrokeColor(Color.BLUE);
 line.setLifelineLength(100);

//Add popup menu to a lifeline segment
 JPopupMenu popup2 = new JPopupMenu();
 segment.setPopup(popup2);
 line.setPopup(popup2);

 //Layout edges
Map<SequenceEdge,ObjectNode[]> edgeNodeMapping = new HashMap<SequenceEdge,ObjectNode[]>();
List<SequenceEdge> allEdges = TestUtil.createSimpleSequnceEdgeAndAssociation(edgeNodeMapping);

DefaultEdgeLayoutStrategy defaultLayout = new DefaultEdgeLayoutStrategy();
//We can modify the edge space
 defaultLayout.setEdgeSpace(20);
 EdgeLayoutStrategy algo = defaultLayout;
 try {
        List<LifelineSegment> lifelineSegments = algo.layout(allEdges,edgeNodeMapping);
 } catch (EdgeLayoutException e) {
        fail("Should not get EdgeLayoutException");
 }
```

5. Future Enhancements

- To make additional rendering schemes
- To add edge line and arrow thickness
- Add more layout algorithms