Visualizer v1 Developer Guide

This documentation is only for Version 1 of Visualizer. For version 2 (Kalileo), follow this link

Outline

Introduction

Visualizer allows Flex/AS3 developers to display a data source as a connected or disconnected graph with a high flexibility level and analyze its structure using our provided layouts. As the need of valuable solutions for data visualization increases, Visualizer will give you an easy, simple and flexible way to meet all your needs for supervising, visualizing and analyzing your data.
Based on our data visualization and data parsing libraries, Visualizer is an innovative Flex component that provides a set of options making easy to control, understand and customize your data visualization. It supports most common data formats (XML, CSV, GML...) and lets integrators choose the way data will be reconstructed and interpreted.
Using our component, it will be possible to extract, analyze and visualize hierarchies, clusters and disconnected entities from any CSV, XML, GML or ICollectionView instances. The generated output can be fully customized, controlled and simplified via automatic Expand/Collapse, coloring and reorganization.

Visualizer Integration

Visualizer can be integrated easily in any Flex application. You only need to:

- import the SWC containing the Visualizer component;
- insert it in the MXML code;
- specify the dataProvider to be visualized;
- specify the dataProvider analysis and interpretation options (analysisPath, Reporting functions...);
- specify other optional parameters (layout, label fields, tooltipFields...).
  When integrating the component, non specified options will be replaced internally by default ones (Default Item Renderer, Default Tooltip...).
  Visualizer output is shown in the following sample. See how Visualizer has interpreted the content of the XML file and created correct content for tooltips and labels while pointing out hierarchy structure using coloring and layout features.

Visualizer can be also integrated in PHP, ASP, JSP and other languages via flash vars that can be associated to a Visualizer wrapper.

Basic Integration sample

- MXML Source :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:KapLab="com.kapit.visualizer.*">
	<mx:Script>
		<![CDATA[
			import mx.controls.Image;
			import
com.kapit.visualizer.LayoutConstants;
		]]>
	</mx:Script>

	<mx:XML xmlns="" id="myDataProvider">
		<Organization name="Manager" experience="20">
			<Organization name="Project Lead"
experience="10">
				<Organization name="Architect"
experience="6"/>
				<Organization name="Developer"
experience="3"/>
```

```
                                <Organization name="Developer"
experience="1"/>
                                <Organization name="Developer"
experience="2"/>
                        </Organization>
                        <Organization name="Project Lead"
experience="8">
                                <Organization name="Architect"
experience="7"/>
                                <Organization name="Architect"
experience="5"/>
                                <Organization name="Developer"
experience="4"/>
                                <Organization name="Developer"
experience="3"/>
                                <Organization name="Developer"
experience="3"/>
                                <Organization name="Developer"
experience="1"/>
                        </Organization>
                        <Organization name="Project Lead"
experience="12">
                                <Organization name="Architect"
experience="7"/>
                                <Organization name="Architect"
experience="7"/>
                                <Organization name="Architect"
experience="5"/>
                                <Organization name="Developer"
experience="3"/>
                                <Organization name="Developer"
experience="3"/>
                                <Organization name="Developer"
experience="2"/>
                                <Organization name="Developer"
experience="1"/>
                        </Organization>
                </Organization>
        </mx:XML>
        <KapLab:Visualizer x="0" y="0" width="800" height="600"
                backgroundColor="0xFFFFFF"
                id="integrationDemo"
                dataProvider="{ myDataProvider}"
                labelFields="{['@name']}"
                tooltipFields="{['@experience']}"
                toolTipTitles="{['Experience']}"
                layout="{LayoutConstants.BALLOON_LAYOUT}"
        />
</mx:Application>
```

- Output :

Data Provider

Visualizer was designed to handle any type of data and display it as a connected or disconnected graph. Using a common internal data parsing model, it enables advanced interpretation of any input data via reconstruction options that can be applied on any data input. Thus, using Visualizer, any data have different meanings. In this section, we will show options that enables data interpretation and reconstruction.
As other KapIT components, Visualizer handles the same data input meaning :

- ICollectionView instances including XML, ArrayCollection...;
- Objects with children property or with a custom TreeDataDescriptor;
- CSV files.

It is advised to visit the common Data Input section of all Kap Lab components.
In this section, we have classified Visualizer options by data input format and we show the power of Kap Lab components when dealing with data reconstruction. We will go beyond standard description of data input to help developers identify the adequate solution for their needs.

ICollectionView data : From simple hierarchies rendering to visual mining

The ICollectionView instances are commonly used to convey Hierarchical data, thus, when used as a data input of Visualizer it will display the data hierarchy. All Classes implementing the ICollectionView interface(XML, XMLListCollection, ArrayCollection...) are treated in the same way.

In the following sample, we use an XML file to display a Organization Hierarchy. We will use the multimediaClass and multimediaDataSource to display corresponding icons for each element instead of rectangles with item names.

Basic XML integration sample

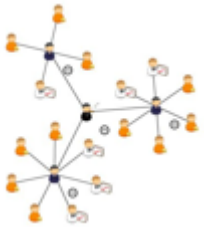
- XML Data source:

```
<mx:XML xmlns="" id="myDataProvider">
     <Organization name="Manager" image="assets/Manager.swf">
        <Organization name="Project Lead" image="assets/ProjectLead.swf">
          <Organization name="Architect" image="assets/Architect.swf"/>
          <Organization name="Developer" image="assets/Developer.swf"/>
          ...
```

- MXML code:

```
<KapVisualizer:Visualizer
backgroundColor="0xFFFFFF"
id="OrganizationChart"
layout="{LayoutConstants.BALLOON}"
dataProvider="{myDataProvider}"
multimediaClass="{Image}"
multimediaDataSource= "@image"
tooltipField="@name"
/>
```

- Output:



In some cases, ICollectionView instances contains hidden information that need to be extracted and analysed before display. Visualizer does this for you. In fact, based on our intelligent data parsing engine, Visusalizer will let developer choose to ignore root node to display disconnected graphs from any ICollectionView instance or/and to analyse the data content based on an Analysis Path and some reporting functions, See Common Data Input Section .
The following sample shows Visualizer ability to interpret hidden relationships inside an XML. A molecule structure is extracted from an abstract representation based on an analysisPath specifying the properties' relationship that we want to show. A labelFunction has been used to deal with nodes labels.

When dealing with XML files (ICollectionView instances in general), make sure that the analysisPath contains a logical relationship and a valid properties.

Advanced mining integration sample

- XML Data source:

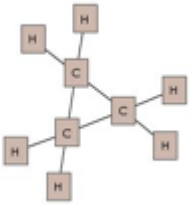
```
<mx:XML xmlns="" id="cycloPropane">
     <Molecule>
              <Molecule atom="C_1" connectedTo="H_1"/>
              <Molecule atom="C_1" connectedTo="H_2"/>
              <Molecule atom="C_1" connectedTo="C_2"/>
              <Molecule atom="C_2" connectedTo="H_3"/>
              <Molecule atom="C_2" connectedTo="H_4"/>
              <Molecule atom="C_2" connectedTo="C_3"/>
              <Molecule atom="C_1" connectedTo="C_3"/>
              <Molecule atom="C_3" connectedTo="H_5"/>
              <Molecule atom="C_3" connectedTo="H_6"/>
     </Molecule>
</mx:XML>
```

MXML code:

```
<mx:Script>
<![CDATA[
public function getLabel(data:Object):Array
{
     if(!data)
              return [""];
     var id:String=data.id;
     if(id.charAt(0)=='C')
              return ["C"];
     else
              return ["H"];
     }
]]>
</mx:Script>
<KapLab:Visualizer id="orgChart" x="0" y="0" width="800" height="600"
              backgroundColor="0xFFFFFF"
              layout="{LayoutConstants.STATIC_ORGANIC_LAYOUT}"
              labelFieldFunction="{getLabel}"
              analysisPath="{['@atom','@connectedTo']}"
```

```
            ignoreRoot="{true}"
            dataProvider="{cycloPropane}"
    />
```

- Output:



When setting the ignoreRoot property to true without specifying the analysisPath, Visualizer will show a disconnected graph where each subGraph corresponds to a direct child of the input. Thus, the ignoreRoot property can be usefull to show disconnected entities, processes or networks.

Objects with children property or with customTreeDataDescriptor

In some applications, it can be interesting to display the content of internal data structures like in case of administration or introspection application. Visualizer allows such functionality : the developper should only give the reference to its object instance and a custom Tree Data Descriptor (via the treeDataDescriptor property) allowing custom navigation inside the object (only if it hasn't the children property).
The options used for ICollectionView instances described previously can be applied to this kind of data format.

CSV Data

CSV files are organized in rows and columns using a CSV delimiter. Records can be delimited by a record Delimiter. A data field has a row index and column index, generally described in a String format while it can represent a Date, a Number or other content. Visualizer handles CSV parsing, reconstruction and data content interpretation via a set of options. In this part we will show how Visualizer meets the needs for CSV content visualization.
Visualizer takes as input:

- ***Required*** analysis path used to extract needed data from the CSV file (analysisPath);
- ***Optional*** CSVDelimiter (**csvDelimiter** by default ";");
- ***Optional*** Record Delimiter (**recordDelimiter** by default "");
- ***Optional*** With Headers (**withHeaders** property by default true);
- ***Optional*** Reporting functions (**reportingFunctions** by default null);
- ***Optional*** Merge Descriptor (**mergeDescriptor** by default null);
- ***Optional*** Attributes Descriptor (**attributesDescriptor** by default null);
  Visualizer is able to add extra CSV parsing and display functionalities by :
- extracting disconnected sets from the parsed CSV;
- indicating Hierarchies (Via Expand/Collapse buttons and Coloring sequences);
- layouting subGraphs;
- Assigning IDs to each element data : In fact, when visualizing a CSV file, Visualizer assigns a constructed data

object to each node. Each data contains :

- ID (**id** property) : extracted given the analysisPath and the columns content ;
- Column Reference (**columnReference** property) : Indicating the column from which the data item is extracted;
- Column Names as Properties : Among data properties, we always find headers names or assigned headers, if no header is in the CSV, properties are column indexes);
  Take a look at CSV Data Handling.
  In the following sample, we expose a simple approach to manipulate CSV data input. Visualizer needs only the *analysisPath* and the *csvDelimiter* to interpret correctly the data input. In this sample, we can remark that the *tooltipFields* and *tooltipTitles* have the same length of the analysisPath, as we make a index based correspondence between column reference in the *analysisPath* and the tooltip array.

It is not mandatory to have the same analysisPath *length for* labelFiels *and* tooltipFields *array. You can specify a unique array that will be used for all generated nodes.*

Simple CSV integration sample

- CSV Data Source :

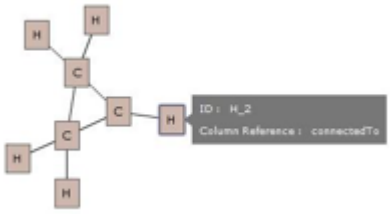
```
atom,connectedTo
C_1,H_2
C_1,C_2
C_2,H_3
C_2,H_4
C_2,C_3
C_1,C_3
C_3,H_5
C_3,H_6
```

MXML Code :

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" xmlns:KapLab="com.kapit.visualizer.*" >
<mx:Script>
<![CDATA[
import mx.controls.Image;
import com.kapit.visualizer.LayoutConstants;

[Embed(source="/csvSource.csv",mimeType="application/octet-stream")]
private var csvThing:Class;
private   var cycloPropane:String;
public function getLabel(data:Object):Array
{
 if(!data)
     return [""];
 var id:String=data.id;
 if(id.charAt(0)=='C')
   return ["C"];
 else
   return ["H"];
}
private function loadData():void
{
 var byteArray:ByteArray;
 var content:String;
 byteArray = new csvThing() as ByteArray;
 cycloPropane=byteArray.readUTFBytes(byteArray.length);
 cyclo.dataProvider=cycloPropane;
}
```

```
]]>
</mx:Script>
<KapLab:Visualizer id="cyclo" x="0" y="0" width="800"
height="600"
        backgroundColor="0xFFFFFF"
        layout="{LayoutConstants.STATIC_ORGANIC_LAYOUT}"
        labelFieldFunction="{getLabel}"
        analysisPath="{['atom','connectedTo']}"

tooltipFields="{[['id','columnReference'],['id','columnReference']]}"
        toolTipTitles="{['ID','Column Reference']}"
        csvDelimiter=","
                creationComplete="loadData()"
/>
</mx:Application>
```

- Output:



Advanced CSV integration sample

- CSV Data Source :

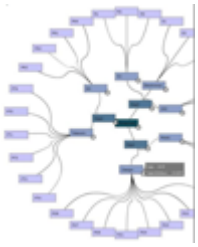
```
Entreprise;Departement;Technology;Name;Age;Work
days;Total days
EntrepriseX;Dep1;Music;PM1;19;40;50
EntrepriseX;Dep1;Music;PM2;20;70;100
EntrepriseX;Dep1;Music;PM3;17;40;55
EntrepriseX;Dep1;Design;PD1;24;12;75
EntrepriseX;Dep1;Design;PD2;23;12;20
EntrepriseX;Dep1;Design;PD3;24;45;80
EntrepriseX;Dep1;Design;PD4;23;12;30
EntrepriseX;Dep1;Design;PD5;24;12;30
EntrepriseX;Dep1;Design;PD6;23;45;50
EntrepriseX;Dep1;Design;PD7;24;80;40
EntrepriseX;Dep1;Design;PD8;23;10;20
EntrepriseX;Dep2;Telecom;PT1;19;10;15
EntrepriseX;Dep2;Telecom;PT2;23;10;14
EntrepriseX;Dep2;Telecom;PT3;19;20;24
EntrepriseX;Dep2;Telecom;PT4;23;20;28
EntrepriseX;Dep2;Telecom;PT5;19;23;26
EntrepriseX;Dep2;Telecom;PT6;23;24;25
EntrepriseX;Dep2;RD;PR1;23;50;55
EntrepriseX;Dep2;RD;PR2;20;40;80
EntrepriseX;Dep2;RD;PR3;23;60;70
EntrepriseX;Dep2;RD;PR4;21;40;50
EntrepriseX;Dep3;PC;M1;23;35;40
EntrepriseX;Dep3;PC;M2;23;40;45
EntrepriseX;Dep3;PC;M3;23;45;50
EntrepriseX;Dep3;Electronics;E1;22;20;25
EntrepriseX;Dep3;Electronics;E2;21;20;30
EntrepriseX;Dep3;Electronics;E3;23;15;20
EntrepriseX;Dep3;IRD;IE1;22;20;30
EntrepriseX;Dep3;IRD;IE2;21;20;25
EntrepriseX;Dep3;IRD;IE3;23;24;30
```

MXML Code :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" xmlns:KapLab="com.kapit.visualizer.*" >
<mx:Script>
<![CDATA[
import fr.kapit.data.descriptor.MergeEntity;
import fr.kapit.data.utils.ReportingUtils;
import fr.kapit.data.descriptor.AttributesEntity;
import fr.kapit.data.descriptor.AttributesDescriptor;
import fr.kapit.data.descriptor.MergeDescriptor;
import mx.controls.Image;
import com.kapit.visualizer.LayoutConstants;

[Embed(source="/csvSource.csv",mimeType="application/octet-stream")]
private var csv:Class;
private  var entrepriseHierarchy:String;
private function mergeFunction(arr:Array):Number
{
  if(arr.length!=2)
   return 0;
  return Number(arr[1])-Number(arr[0]);
}
private function loadData():void
{
 var byteArray:ByteArray;
 var content:String;
 byteArray = new csv() as ByteArray;

entrepriseHierarchy=byteArray.readUTFBytes(byteArray.leng
th);
 var mergeDescriptor:MergeDescriptor=new
MergeDescriptor();
 var mergeEntity:MergeEntity=new MergeEntity(["Work days","Total
days"],"Holidays",mergeFunction);
 mergeDescriptor.addMergeDescription(mergeEntity);
 mergeDescriptor.leaveMergeColumns=true;
 var attributesDescriptor:AttributesDescriptor=new
AttributesDescriptor();
 var entrepriseAttributesEntity:AttributesEntity=new
AttributesEntity("Entreprise",["Holidays","Work days","Age"]);
 var nameAttributesEntity:AttributesEntity=new
AttributesEntity("Name",["Holidays","Age"]);
 var depAttributesEntity:AttributesEntity=new
AttributesEntity("Departement",["Holidays","Total days","Age"]);

attributesDescriptor.AddAttributesDescription(entrepriseA
ttributesEntity);

attributesDescriptor.AddAttributesDescription(nameAttribu
tesEntity);

attributesDescriptor.AddAttributesDescription(depAttribut
esEntity);
 var reportingFunctions:Dictionary=new Dictionary();
 reportingFunctions["Age"]=ReportingUtils.mean;
 reportingFunctions["Holidays"]=ReportingUtils.mean;
 reportingFunctions["Total days"]=ReportingUtils.mean;
 reportingFunctions["Work days"]=ReportingUtils.mean;
 entreprise.dataProvider=entrepriseHierarchy;
 entreprise.csvDelimiter=";";
```

```
entreprise.analysisPath=["Entreprise","Departement","Technology","Name
"];
 entreprise.withHeaders=true;
 entreprise.mergeDescriptor=mergeDescriptor;
 entreprise.reportingFunctions=reportingFunctions;

entreprise.labelFields=[["Entreprise"],["Departement"],["Technology"]
,["Name"]];

entreprise.coloringPolicy=Visualizer.BICHROMATIC_COLORING
;

entreprise.tooltipFields=[["Age"],["Age","Holidays"],["Age","Holiday
s"],["Age","Holidays"]];

entreprise.toolTipTitles=[["Age"],["Age","Holidays"],["Age","Holiday
s"],["Age","Holidays"]];
}
]]>
</mx:Script>
<KapLab:Visualizer id="entreprise" x="0" y="0" width="1000"
height="900"
        backgroundColor="0xFFFFFF"
        layout="{LayoutConstants.RADIAL_LAYOUT}"
                creationComplete="loadData()"
/>
</mx:Application>
```

- Output:



GraphML Data

GML (GraphML) format is an XML based structure that describes graphs using *<node>* and *<edge>* tags (see GraphML primerhttp://graphml.graphdrawing.org/primer/graphml-primer.html). The actual version of GML parsing **doesn't support advanced concepts of GraphML (SubGraphs, HyperEdges, SVG support ...)**, but this feature will be handled in next versions according to Visualizer roadmap. In the other hand, overriding GraphML definitions with some custom node attributes or edge attributes is possible.
In this part we will show how Visualizer meets needs for GML content visualization.
For GraphML display, Visualizer takes as input a GML file in an **XML** data type format.

The GML file must contain the surrounding <graphml .... /graphml> tag in order to let Visualizer autoimatically detect that the XML input is a GML. Thus the processing should be done according to GraphML specifications.

Visualizer is able to add extra Graph ML parsing and display functionalities by :

- extracting disconnected sets from the parsed GML;

- indicating Hierarchies (Via Expand/Collapse buttons and Coloring sequences);
- layouting subGraphs;
- Assigning IDs to each element data according to the id attribute of each GraphML element.

GML integration sample

GML Data Source :

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <node id="n4"/>
    <node id="n5"/>
    <node id="n6"/>
    <node id="n7"/>
    <node id="n8"/>
    <node id="n9"/>
    <node id="n10"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
    <edge source="n3" target="n5"/>
    <edge source="n3" target="n4"/>
    <edge source="n4" target="n6"/>
    <edge source="n6" target="n5"/>
    <edge source="n5" target="n7"/>
    <edge source="n6" target="n8"/>
    <edge source="n8" target="n7"/>
    <edge source="n8" target="n9"/>
    <edge source="n8" target="n10"/>
  </graph>
</graphml>
```

MXML Code :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:visualizer="com.kapit.visualizer.*" viewSourceURL="srcview/index.html">
     <mx:Script>
             <![CDATA[
                     import com.kapit.visualizer.events.VisualizerEvent;
                     import com.kapit.visualizer.LayoutConstants;

                     private function onCreationComplete():void
                     {
                             var request:URLRequest = new
URLRequest("gmlfile.xml");
                             var loader:URLLoader = new
URLLoader(request);
                             loader.addEventListener(Event.COMPLETE,
setDataprovider);
                     }
```

```
                    private function setDataprovider(event:Event):void
                    {
                            var gmlData:XML;
                            gmlData = new XML((event.currentTarget as
URLLoader).data);
                            visualizer.dataProvider = gmlData;

     visualizer.addEventListener(VisualizerEvent.ELEMENTS_EXPANDED_CO
LLAPSED, elementsExpanded)
                    }

                    private function
elementsExpanded(event:VisualizerEvent):void
                    {

                    }
             ]]>
     </mx:Script>
     <visualizer:Visualizer
                 backgroundColor="0xffffff"
             id="visualizer"
             width="100%"
             height="100%"
             layout="{LayoutConstants.STATIC_ORGANIC_LAYOUT}"
             creationComplete="onCreationComplete()"/>
</mx:Application>
```

Visualizer output:

User Driven Data Input

Visualizer can handle user driven data input by exposing several methods to let developers feed Visualizer with content without having to use the**dataProvider** property.
These methods supports adding/removing nodes and links and gives the ability to define programmatically nodes children when adding them. Among these methods :

- **addNodeElement** : Adds a Data as Visualizer Node to the Graph with respect to Expand/Collapse policy as the developer can specify a parent Node. If data is GML, the infomation should be given to this function
- **addLinkElement** : Adds a data as Visualizer link to the graph given its source and target nodes and optionally its style.
- **removeNodeElement**: Removes a node given its unique ID. You can specify if children should be removed to
- **removeLinkElement** : Removes a link given its unique ID
- **removeAll** : Removes all Visualizer graph.

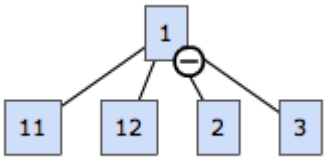
Be carefull that nodes rendering properties and link rendering properties are compatible to the data that you provide to Visualizer.

User Driven Data Input integration sample

- MXML Code :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:KapLab="com.kapit.visualizer.*">
	<mx:Script>
		<![CDATA[
			import mx.collections.ArrayCollection;
			import
com.kapit.visualizer.renderer.GenericObject;
			import
com.kapit.visualizer.renderer.GenericSprite;
			import mx.controls.Label;
			import mx.containers.Canvas;
			import mx.core.UIComponent;
			import
com.kapit.visualizer.renderer.GenericLink;
			import mx.controls.Image;
			import
com.kapit.visualizer.LayoutConstants;

			private function onCreationComplete():void
			{
				var center:Point = new
Point(myVisualizer.width/2, myVisualizer.height/2);
				var node1:GenericSprite =
myVisualizer.addNodeElement({id:'1'},null,center);
				node1.type =
GenericObject.EXPAND_SPRITE;
				var node2:GenericSprite =
myVisualizer.addNodeElement({id:'11'},node1,center);
				var node3:GenericSprite =
myVisualizer.addNodeElement({id:'12'},node1,center);
				var node4:GenericSprite =
myVisualizer.addNodeElement({id:'2'},null,center);
				var node5:GenericSprite =
myVisualizer.addNodeElement({id:'3'},null,center);

	myVisualizer.addLinkElement({id:'l1'},node1,node2);

	myVisualizer.addLinkElement({id:'l2'},node1,node3);

	myVisualizer.addLinkElement({id:'l3'},node1,node4);

	myVisualizer.addLinkElement({id:'l4'},node1,node5);
				myVisualizer.reLayout();
			}
		]]>
	</mx:Script>

	<KapLab:Visualizer x="0" y="0" width="100%" height="100%"
		backgroundColor="0xFFFFFF"
		id="myVisualizer"
		labelFields="{['id']}"
		coloringPolicy="{Visualizer.UNIFORM_COLORING}"
		layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"
		creationComplete="{onCreationComplete()}"
	/>
</mx:Application>
```

- Output :



Analysis Path

Analysis Path is the most important concept when analyzing information encapsulated and hidden inside any data. It defines what we want to see and how links must be constructed. An analysis path is an array containing a selection of Headers (for CSV) or Node attributes (for XML or GML files) or Object properties (for objects with children or customTreeDataDescritor) in a given order that defines the linkage policy.
When defining an analysis path, the input data will be transformed into an internal data structure that handles any manipulation or analysis task.
To illustrate the importance and the usage simplicity of analysisPath, we give a usage example for a CSV file (the example can be ported to other data formats). We assume having an Enterprise Organization encapsulated in the following CSV file.

CSV Data Source

```
Enterprise;Department;Qualification;Name;Age
EnterpriseX;Management;Manager; M1; 45
EnterpriseX;Management;Finance Head; M2; 50
EnterpriseX;Management;Production Head; M4; 45
EnterpriseX;Management;Research & Developement Head; M3; 35
EnterpriseX;Finance; Finance  Team Leader; FT1; 45
EnterpriseX;Finance;Supervisor; FSP1; 35
EnterpriseX;Finance;Finance Engineer; FE1; 30
EnterpriseX;Finance;Finance Engineer; FE2; 34
EnterpriseX;Finance;Secretary; FS1; 25
EnterpriseX; Production; Production Team Leader; PT1; 45
EnterpriseX; Production;Quality Engineer; PQ1; 35
EnterpriseX; Production;Production Engineer; PE1; 25
EnterpriseX; Production;Production Engineer; PE2; 30
EnterpriseX; Production;Production Engineer; PE3; 28
EnterpriseX; Production;Production Engineer; PE4; 29
EnterpriseX; Production;Production Engineer; PE5; 30
EnterpriseX; Production;Production Engineer; PE6; 34
EnterpriseX; Production;Production Secretary; PS1; 25
.....
```

The analysisPath can be set according to the developer needs. If we want to show Enterprise Human Resources Qualifications and to classify employees according to that, the analysisPath should be

```
["Enterprise","Qualification","Department"]
```

In case where more filtering is needed or where departments must be visualized, the analysisPath should be :

```
["Enterprise","Department","Qualification","Department"]
```

The analysisPath property can be used combined to reportingFunctions defining the way data reconstruction is done and other options that have been discussed in the data provider section.

Analysis Path should be set to null or updated if data provider format changes.

If we used an analysis path for a given visualizer data provider and then we change the data provider format or type or structure for another analysis and display task, the analysis path should be:

- set to null : if the analysis is Hierarchical (with ot without ignoreRoot option);
- updated : if the data provider has a different structure that must be reconstructed and analysed.

***Related Visualizer properties**: analysisPath, reportingFunctions, mergeDescriptor, attributesDescriptor, customTreeDataDescriptor, typesDescriptor.*
For more information read Advanced CSV properties

Visualizer Renderering

Node Rendering

Node rendering refers to the way Graph Nodes are populated with a specific content extracted from the data that it represents. When using Visualizer, nodes content can be defined using :

- The DefaultItemRenderer loaded by default and taking as parameters visualizer properties related to labeling and coloring (labelFields,labelTitles, firstColor, secondColor..);
- Combination between a UIComponent Class having a data property or Image class (via multimediaClass property) and the data extraction schema which is not mendatory(multimediaDataSource). The given renderer Class will be initialized for each node and will receive as data :
    - Whole node data if the multimediaDataSource property is set to null;
    - Value of node data property. The property is given by the user in the multimediaDataSource property.
- Custom ItemRenderer generation function that takes as input the node data, processes the data and returns the required DisplayObject instance. This function can be also used to define extra options and customization of the DefaultItemRenderer.

DefaultItemRenderer and automatic rendering

The Item renderer is by default set to DefaultItemRenderer, a renderer implementing the IListCollection interface. The

content of nodes are defined via several Visualizer properties classed as follows:

Labels properties

- **Optional** labelFields : an array indicating the fields in the data input that must be displayed inside each node. If null, the nodes show their IDs as Labels :
  - for an XML input, it must be XML Element attributes like "@name" and "@age" (provided as an Array[visualizer:'@name','@age'];
  - for a CSV file, it must be some input headers (if no headers are defined, an Array of column indexes should be provided) and should be a two levelled array as for each field in the analysisPath a label Array is defined. A labelFields for a given CSV with an *analysisPath* length equal to 3 should have the same structure as [visualizer:[X,Y],[visualizer:Z,X],[W]].
- **Optional** labelTitles : defines title for each label field in the *labelFields* array. If not provided, nodes content is filled only with extracted labels content.
- **Optional** labelFieldFunction : defines a function for label content extraction. The function takes as input a data object and returns an array of extracted labels (array of strings or numbers) that must be displayed inside each node. When using thelabelFieldFunction, the labelTitles can be also set as the content of the function output is known by the developer.
  The following sample show how we can use **labelFields** or **labelFieldsFunction** to show each employee node its name, age and qualification.

- XML Data Source :

```
<KapIT EmployeeName=XXX EmployeeAge=XXX
EmployeeQualification=XXX>
...
 ...
  ...
</KapIT>
```

MXML Code :

```
<KapLab:Visualizer
                  id="orgChart" x="0" y="0" width="800" height="600"
               backgroundColor="0xFFFFFF"
               layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"

      labelFields="{['@EmployeeName','@EmployeeAge','@EmployeeQualification']}"
               labelTitles="{['Name','Age','Qualification']}"
               dataProvider="{myXML}"
               firstColor="0x334455"
               secondColor="0x554433"
               coloringPolicy="{Visualizer.BICHROMATIC_COLORING}"
      />
```

LabelFieldFunction :

```
private function customLabelFunction (data:Object):Array
{
  if(!data)
     return null;
  return
[data["@EmployeeName"],data["@EmployeeAge"],data["@EmployeeQualification"]];
}
```

Coloring properties

- **Optional** coloringPolicy : a String that defines a coloring policy in Visualizer and that are restricted to :
  - Bichromatic coloring (BICHROMATIC_COLORING) : defining a bichromatic coloring for

hierachies only and that uses *firstColor* and *secondColor* properties to color levels from top to bottom with respect of bichromatic restrictions;

- Monochromatic coloring (MONOCHROMATIC_COLORING): defining a bichromatic coloring for hierachies only and that from *firstColor* to black color;
- Uniform coloring (UNIFORM_COLORING) : defining a uniform node coloring based on the content of the static color field uniformColor.

- **Optional** **colorFields** : an Array of colors, each one is affected to elements created from a column or attribute reference of theanalysisPath. ColorFields are taken into account in case where the generated output isn't a tree. (Example: analysisPath=[visualizer:"atom","connectedTo"] -> colorFields=[visualizer:0x445566, 0x665544]);
- **Optional** **coloringFunction** : a function that takes as input a data object (a data set from the data provider having known properties and refering to a given node) and returns its color given some specific developer considerations.
- **Optional** **firstColor** : first extremity of a bichromatic coloring policy;
- **Optional** **secondColor** : first extremity of a bichromatic coloring policy;
- **Optional** **uniformColor** : used color of monochromatic coloring policy;

The following sample describes how we can apply a coloring function on Visualizer nodes to differentiate some nodes from others according to their type. We use a **coloringFunction** to achieve this requirement.

- CSV Data Source :

```
EntrepriseName,EntrepriseDepartment,EntrepriseEmployeeID,EntrepriseEmployeeName,MoneyOutput
...
...
...
```

MXML code :

```
<mx:Script>
<![CDATA[
public function getColor(data:Object):Array
{
       if(!data)
                return [""];
         trace(data.id)  //*just showing the id property which is important when analysing*
                                // *cyclic graphs in CSV*
       var columnReference:String=data.columnReference;
       if(id.charAt(0)=='EntrepriseName')
                return 0x112233;
         else if(id.charAt(0)=='EntrepriseDepartment')
                   return 0x445566;
       else
                return 0x778899;
}
]]>
</mx:Script>
<KapLab:Visualizer
                   id="companyStructure" x="0" y="0" width="800" height="600"
                backgroundColor="0xFFFFFF"
                   dataProvider="myCSVr"
                layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"
                labelFields="{[['EntrepriseName','MoneyOutput'],
['EntrepriseDepartment','MoneyOutput'] ,
['@EntrepriseEmployeeName']]}"
                   coloringFunction="{getColor}"
```

```
/>
```

Font properties

??The font properties of the DefaultItemRenderer can be also be controlled via several properties like labelFontColor, labelFontFamily labelTitleFontColor and labelTitleFontFamily??

Multimedia class and direct rendering

Visualizer enables developer to set a different item renderer instead of the DefaultItemRenderer by providing:

- **Required** the IListItemRenderer class (having the data property) or an Image class via the **multimediaClass** property;
- **Optional** the field from which data is extracted and provided to the internally created custom item renderer via **multimediaDataSource**.

Multimedia Class integration sample

- XML Data source:

```
<mx:XML xmlns="" id="myDataProvider">
       <Organization name="Manager" image="assets/im1.swf">
          <Organization name="Project Lead" image="assets/im21.swf">
            <Organization name="Architect" image="assets/im22.swf"/>
            <Organization name="Developer" image="assets/im23.swf"/>
            ...
```

- MXML code:

```
<KapVisualizer:Visualizer
id="myVisualizer"
layout="{LayoutConstants.RADIAL}"
dataProvider="{myDataProvider}"
multimediaClass="{Image}"
multimediaDataSource= "@image"/>
```

Custom Node rendering

Using the **multimediaFunction** property, it is possible to have different content for each node as this function return for each node the adequate IListItemRenderer which can be a (Video, custom Renderer ...).
The following sample describes how given an XML structure we are able to renderer different UIComponents for visualizer nodes using the**multimediaFunction** that returns a DisplayObject instance. Our need is to show each employee node its name, age and qualification.

MultimediaFunction integration sample

- XML Data Source :

```
<KapIT EmployeeName=XXX EmployeeAge=XXX EmployeeQualification=XXX>
...
  ...
    ...
<KapIT/>
```

- Multimedia Function :

```
private function customItemRendererFunction
(data:Object):IListItemRenderer
{
  if(!data)
```

```
    return null;
  myContent:SomeComponent=new SomeComponent();
  myContent.data=data;
  if(data['@ EmployeeAge']<20)
    myContent.color=0x555555;
  return myContent;
}
```

Link Rendering

Links rendering can be customized in Visualizer using some advanced features combined to styling properties. Despite the possibility of configuring how Links are styled (see visualizer:Links style section), links can show some extra data via labels over the links or even some custom components.
The feature of links decorating is applicable only in the case where links has data. Thus this feature is availbale only for GML input or user driven data input (User Driven Data Input).
Like nodes rendering, link decorators can be integrated via :

- DefaultItemRenderer (By Default) : places a standard String (extracted from the link data given corresponding properties) over the Link . It is a combination of :
    - **REQUIRED** **labelLinkFields** : an array indicating the fields in the link data (ex : gml edge properties) that must be displayed above each link;
    - **REQUIRED** **labelLinkFieldFunction** : required if no labelLinkFields is defined. defines a function for label content extraction. The function takes as input a data object and returns an array of extracted labels (array of strings or numbers) that must be displayed inside each link decorator;
    - **OPTIONAL** **labelLinkTitles** : defines title for each label field in the labelLinkFields array. If not provided, links decorator content is filled only with extracted labels content.

The font properties of the DefaultItemRenderer can be also be controlled via several properties like labelLinkFontColor, labelLinkFontFamily or labelLinkTitleFontColor and labelLinkTitleFontFamily

- Multimedia class and direct rendering : defines a custom link decorator class (an UIComponent) that will be placed above the link. Like nodes, it requires :
    - **REQUIRED** the IListItemRenderer class (having the data property) or an Image class via the **multimediaLinkClass** property;
    - **OPTIONAL** the field from which data is extracted and provided to the internally created custom item renderer via **multimediaLinkDataSource**. If null, the decorator will be initiailized with the link data.
- Custom rendering function **multimediaLinkFunction** : takes as an input a data object and returns a given UIComponent according to developer needs.
  In the following samples, we show how easy and flexible to show link decorators with Visualizer. Decorators can be highly important to vehiculate a complete understanding of the input data.

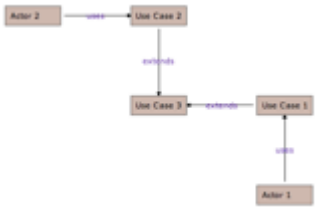
Default Link Decorator integration sample

- MXML Code :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:KapLab="com.kapit.visualizer.*">
     <mx:Script>
            <![CDATA[
                    import mx.controls.Label;
                    import mx.containers.Canvas;
```

```
                    import mx.core.UIComponent;
                    import com.kapit.visualizer.renderer.GenericLink;
                    import mx.controls.Image;
                    import com.kapit.visualizer.LayoutConstants;

                    [Bindable]
                    private var linkStyle:Object =
                    {
                            arrowPolicy:"double",

    arrowSourceType:GenericLink.LINK_ARROW_NONE,

    arrowTargetType:GenericLink.LINK_ARROW_STANDARD_TYPE,

    decoratorPlacement:GenericLink.LINK_DECORATOR_AT_CENTER_PLACEMEN
T
                    }
                    private function onCreationComplete():void
                    {

    myVisualizer.orthogonalLayout.nodesSpacing = 100;
                    }

            ]]>
    </mx:Script>
    <mx:XML xmlns="" id="myDataProvider">
            <graphml>
                    <node id='Actor 1'/>
                    <node id='Actor 2'/>
                    <node id='Use Case 1'/>
                    <node id='Use Case 2'/>
                    <node id='Use Case 3'/>
                    <edge id='1' label='uses' source='Actor 1'
target='Use Case 1'/>
                    <edge id='2' label='uses' source='Actor 2'
target='Use Case 2'/>
                    <edge id='3' label='extends' source='Use Case 1'
target='Use Case 3'/>
                    <edge id='4' label='extends' source='Use Case 2'
target='Use Case 3'/>
            </graphml>

    </mx:XML>

    <KapLab:Visualizer x="0" y="0" width="100%" height="100%"
            backgroundColor="0xFFFFFF"
            id="myVisualizer"
            dataProvider="{ myDataProvider}"
            labelFields="{['id']}"
            linkStyle="{linkStyle}"
            labelLinkFields="{['label']}"
            labelLinkFontColor="0x6333a5"
            layout="{LayoutConstants.ORTHOGONAL_LAYOUT}"
            creationComplete="{onCreationComplete()}"
    />
</mx:Application>
```

- Output :

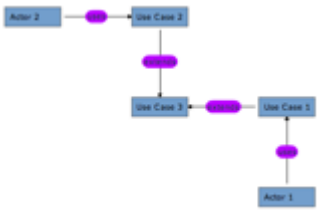


Advanced Link Decorator integration sample

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:KapLab="com.kapit.visualizer.*">
        <mx:Script>
                <![CDATA[
                        import mx.controls.Label;
                        import mx.containers.Canvas;
                        import mx.core.UIComponent;
                        import com.kapit.visualizer.renderer.GenericLink;
                        import mx.controls.Image;
                        import com.kapit.visualizer.LayoutConstants;

                        [Bindable]
                        private var linkStyle:Object =
                        {
                                arrowPolicy:"double",

        arrowSourceType:GenericLink.LINK_ARROW_NONE,

        arrowTargetType:GenericLink.LINK_ARROW_STANDARD_TYPE,

        decoratorPlacement:GenericLink.LINK_DECORATOR_AT_CENTER_PLACEMEN
T
                        }

                        private function onCreationComplete():void
                        {

        myVisualizer.orthogonalLayout.nodesSpacing = 100;
                        }

                        private function
linkDecoratorRenderer(data:Object):UIComponent
                        {
                                if(!data)
                                        return null;
                                var canvas:Canvas = new Canvas();

        canvas.setStyle('backgroundColor',0xba00ff);
                                canvas.setStyle('backgroundAlpha',1);
                                canvas.setStyle('cornerRadius',10);
                                canvas.setStyle('borderStyle','solid');
                                canvas.setStyle('borderColor',0xba00ff);
                                var label:Label = new Label();
                                label.percentHeight=100;
                                label.percentWidth=100;
                                canvas.addChild(label);
                                label.text = data.label;
                                label.setStyle('textAlign','center');
                                label.validateNow();
                                canvas.width =
```

```
label.getExplicitOrMeasuredWidth();
                                canvas.height =
label.getExplicitOrMeasuredHeight();
                                return canvas;
                        }

                ]]>
        </mx:Script>

        <mx:XML xmlns="" id="myDataProvider">
                <graphml>
                        <node id='Actor 1'/>
                        <node id='Actor 2'/>
                        <node id='Use Case 1'/>
                        <node id='Use Case 2'/>
                        <node id='Use Case 3'/>
                        <edge id='1' label='uses' source='Actor 1'
target='Use Case 1'/>
                        <edge id='2' label='uses' source='Actor 2'
target='Use Case 2'/>
                        <edge id='3' label='extends' source='Use Case 1'
target='Use Case 3'/>
                        <edge id='4' label='extends' source='Use Case 2'
target='Use Case 3'/>
                </graphml>

        </mx:XML>

        <KapLab:Visualizer x="0" y="0" width="100%" height="100%"
                backgroundColor="0xFFFFFF"
                id="myVisualizer"
                dataProvider="{ myDataProvider}"
                labelFields="{['id']}"
                linkStyle="{linkStyle}"
                multimediaLinkFunction="{linkDecoratorRenderer}"
                labelLinkFontColor="0x6333a5"
                layout="{LayoutConstants.ORTHOGONAL_LAYOUT}"
                uniformColor="0x729ecc"
                creationComplete="{onCreationComplete()}"
        >
        </KapLab:Visualizer>
</mx:Application>
```

- Output :



Tooltips Rendering

Tooltips are very useful for showing extra information of a node data. Thus, in Visualizer, tooltips content is handled in the same way that node content is handled. Tooltip management is done using :

- DefaultTooltip : loaded by default on *TOOLTIP_SHOW* event and taking as parameters visualizer properties related to tooltip :
  - **REQUIRED** tooltipFields;
  - **REQUIRED** **tooltipFieldFunction** : required if no tooltipFields is defined. defines a function for tooltip content extraction. The function takes as input a data object and returns an array of extracted labels (array of strings or numbers) that must be displayed inside the tootlip;
  - **OPTIONAL** tooltipTitles).
- Custom Tooltip function **tooltipRendererFunction** : takes as input the tooltip event, processes the data of the target (which is a Visualizer node) and sets the event *tootltip* to a given ITooltip instance.

The same tooltip rendering logic is applicable on Link tooltips. The used properties changes. You need just to add "Link" after tooltip for each property.

DefaultTootlip and automatic rendering

The tooltip is by default set to DefaultTooltip, a renderer implementing the ITooltip interface. its content is defined by some Visualizer properties refering to tootlip which are:

- **tooltipFields** : an array indicating the fields in the data input that must be displayed in the tooltip. (same logic as labelFields for all data input)
- **tooltipTitles** : title for each tooltip field in the *tooltipFields* array. If not provided, nodes content is filled only with extracted tooltip content.
- **toolTipFieldFunction** : defines a function for tooltip content extraction instead of tooltipFields. The function takes as input a data object and returns an array of extracted content (array of strings or numbers) that must be displayed. When using the *tooltipFieldFunction*, the*tooltipTitles* can be also set, because the content of the function output is known by the developer
  The following sample shows how **tooltipFields** and **toolTipFieldFunction** can be used to show a wanted tooltip from node data.

  Default Tooltip integration sample for XML

  - XML Data source :

```
// Data source used as data provider for visualizer
<KapIT EmployeeName=XXX EmployeeAge=XXX
EmployeeQualification=XXX>
...
 ...
  ...
</KapIT>
```

  - MXML Code:

```
<KapLab:Visualizer
                id="orgChart" x="0" y="0" width="800" height="600"
              backgroundColor="0xFFFFFF"
              layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"

              labelFields="{['@EmployeeName']}"
                tooltipFields="{['@EmployeeAge','@EmployeeQualification']}"
                tooltipTitles="{['Age','Qualification']}"
              dataProvider="{myXML}"
      />
```

  - Tooltip Function (can be used instead of tooltipFields for more customization) :

```
private function customTootlipFieldFunction (data:Object):Array
```

```
{
  if(!data)
     return  null;
  if(data["@EmployeeAge"]<20)
     return  [data["@EmployeeQualification"]];
  return  [data["@EmployeeAge"],data["@EmployeeQualification"]];
}
```

- CSV Data Source :

```
EntrepriseName,EntrepriseDepartment,EntrepriseEmployeeID,Entrep
riseEmployeeName,MoneyOutput
...
...
...
```

- MXML Code:

```
<KapLab:Visualizer
                 id="companyStructure" x="0" y="0" width="800" height="600"
               backgroundColor="0xFFFFFF"
               layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"
               labelFields="{[['EntrepriseName'],
['EntrepriseDepartment'] , ['@EntrepriseEmployeeName']]}"

tooltipFields=[['moneyOutput'],['moneyOutput'],['moneyOutput']]
                 tootlipTitles=['Money Output']
                 dataProvider="myCSV"
/>
```

Custom Tootlip rendering

Using the **tooltipRendererFunction** property, it is possible to have different content for each node tooltip as this function sets the adequate ITooltip at each tooltip event.
The following sample shows how we used the **tooltipRendererFunction** property to show for each employee node its age and qualification in a custom Tooltip implementing the ITooltip interface and having a data property that processes internally an array of strings.

Custom Tooltip integration sample

- XML Data Source :

```
// Data source used as data provider for visualizer
<KapIT EmployeeName=XXX EmployeeAge=XXX EmployeeQualification=XXX>
...
 ...
  ...
</KapIT>
```

- MXML Code:

```
<mx:Script>
private function tooltipFunction(event:ToolTipEvent):void
{
    var myTooltip:MyToolTip=new MyToolTip(); //MyToooltip implements ITooltip
    var data:Object=event.target.data;
    myTooltip.data=['@EmployeeAge','@EmployeeQualification' ];
    event.tooltip=myTooltip
}
```

```
<KapLab:Visualizer
                id="orgChart" x="0" y="0" width="800" height="600"
            backgroundColor="0xFFFFFF"
            layout="{LayoutConstants.HIERARCHICAL_TREE_LAYOUT}"
            labelFields="{['@EmployeeName']}"
                tooltipRendererFunction="{tooltipFunction}"
            dataProvider="{myXML}"
     />
```

Styling

Link Styling

Links can be styled given a Style Object containing several properties (color, thickness, alpha, dashed or not, extremities rendering .....). By setting the**idleLinkStyle** property, links are re-styled given information in the provided style object.
The properties of a link style object are :

- Standard properties : pixelHinting(false),scaleMode("normal"),caps("none") and joints(null),miterLimit(3);
- thickeness : Link thickeness (1 by default);
- color: Link color (0x000000 by default);
- alpha: Link alpha (0.6 by default);
- renderingPolicy: Indicates if the rendering is dashed or not (values are 'solid' or 'dash', 'solid' value is used by default);
- dashed : Indicates if the rendering is dashed or not (@Depricated);
- onLength : Visible dash length (5 by default);
- offLength:Invisible dash length (5 by default);
- useClipping: Indicates if links extremities should be pointing to the node extremities center or bounded to their exterior border (false by default);
- arrowPolicy: Indicates where arrows should be draw ("none" by default). Values are :
    - "single" : Arrow at target;
    - "double" : Arrow at source and target;
    - "none" : No arrows at extremities;
- arrowWidth: arrow width (7 by default);
- arrowHeight:arrow height (5 by default);
- arrowRadius: arrow radius. Applicable only if is circle typed (5 by default);
- arrowSourceType: Type of arrow at source. It can be circular ("circle"), rectangular ("rectangle") or standard arrow ("standard) ("circle" by default);
- arrowTargetType: Type of arrow at target;
- decoratorPlacement : Position of the Link decorator. It can be at target ("target"), source("source") or center ("center");

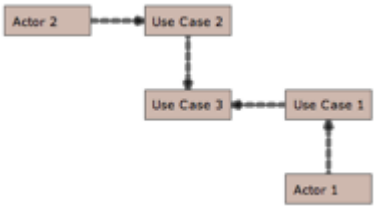
  The behaviour and state of Visualizer links can be also styled by providing a link style object that describes the link rendering at roll over (via **highlightLinkStyle**) or click (via **clickLinkStyle**).
  The link format can be also customized using Layout Link Properties (Orthogonal, Curved and Straight). To use them, you have to set the**edgeDrawing** property (if existing) for the current used layout.
  Link Style integration sample

- Code :

```
?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:KapLab="com.kapit.visualizer.*">
	<mx:Script>
		<![CDATA[
			import mx.controls.Label;
			import mx.containers.Canvas;
			import mx.core.UIComponent;
			import
com.kapit.visualizer.renderer.GenericLink;
			import mx.controls.Image;
			import
com.kapit.visualizer.LayoutConstants;

			[Bindable]
			 private var myStyle:Object={
			thickness:5,
			color:0x000000,
		       alpha:0.6,
		       pixelHinting:false,
		       scaleMode:"normal",
		       caps:"none",
		       joints:null,
		       miterLimit:3,
			    renderingPolicy:"dash",
			    dashed:true,
			    onLength:5,
			    offLength:5,
			    useClipping:false,
			    arrowPolicy:"single",
			    arrowWidth:7,
			    arrowHeight:5,
			    arrowRadius:5,
			    arrowSourceType:"circle",
			    arrowTargetType:"standard"
			    }

		private function onCreationComplete():void
		{
		  myVisualizer.orthogonalLayout.nodesSpacing=50;
		}



		]]>
	</mx:Script>

	<mx:XML xmlns="" id="myDataProvider">
		<graphml>
			<node id='Actor 1'/>
			<node id='Actor 2'/>
			<node id='Use Case 1'/>
			<node id='Use Case 2'/>
			<node id='Use Case 3'/>
			<edge id='1' source='Actor 1' target='Use
Case 1'/>
			<edge id='2' source='Actor 2' target='Use
Case 2'/>
			<edge id='3' source='Use Case 1'
```

```
target='Use Case 3'/>
                        <edge id='4' source='Use Case 2'
target='Use Case 3'/>
                </graphml>
        </mx:XML>


        <KapLab:Visualizer x="0" y="0"
                initialize="onCreationComplete()"
            backgroundColor="0xFFFFFF"
            id="myVisualizer"
            layout="{LayoutConstants.ORTHOGONAL_LAYOUT}"
            dataProvider="{myDataProvider}"
            labelFields="{['id']}"
            idleLinkStyle="{myStyle}"
            width="500"
            height="400"
                />
</mx:Application>
```

- Output :



Node Styling

Nodes can be styled using flex filters. In fact, in idle, roll over or click status, nodes have default filters that are loaded internally. The developer can choose to set his own filters using :

- **idleNodeFilers** : filters at Idle status;
- **highlightNodeFilters** : filters at Roll Over status;
- **clickNodeFilters** : filters at Click status.
  The usage of States, animation and Data Binding are also handled in Visualizer by combining custom item renderers and multimediaFunction property.

Highlight Policies

Nodes highlight can be customized inside Visualizer. In fact, Visualizer offers two highlight policies, each one suited to a given context, via the **highlightPolicy** property:

- **Filter based highlight** (Visualizer.FILTER_BASED_HIGHLIGHT) : default highlight strategy based on adding Filters to Nodes at Roll Over (see **highlightNodeFilters**. This policy is suited for rectangular shapes and for general shapes.
- **Circular based highlight** (Visualizer.CICULAR_BASED_HIGHLIGHT) : This highlight policy highlights nodes with a surrounding circle with white background. This policy is suited for nodes that fits approximately into a

square.

Graph Layout

Based on our Kap Layouts AS3 library, a Kap Lab library that targets the graph drawing and data visualization fields, we have been able to give the possibility to visualize data in different configurations or **layouts** with a high flexibility and customization level. Thus, the scope of use of this component is enlarged to multiple fields (Networking, Chemistry, Hierarchies browsing, Social Networking applications, Data Mining, Process Modeling).
For detailed overview of the available layouts : Kap Layouts
Our Provided layouts are fully customizable via a set of options that controls compactness, angular resolution, orientations, link paths...;
All graph layout algorithms are shared by the component in order to customize the output of any layout and any changes on its properties are automatically applied on Visualizer.
The layout of graphs can be modifiyed by setting the **layout** property to a Layout index. Use **LayoutConstants** class to access to the list of layout reference constants :

- Radial layout : **RADIAL_LAYOUT**;
- Balloon layout : **BALLOON_LAYOUT**;
- Tree layout : **HIERARCHICAL_TREE_LAYOUT**;
- Cyclic Flow layout : **HIERARCHICAL_CYCLIC_LAYOUT**;
- Organic layout : **STATIC_ORGANIC_LAYOUT**;
- Orthognal layout : **ORTHOGNAL_LAYOUT**;
- Layout contstants property values.
  The following sample shows how modifying programmatically default Visualizer layout options:

Graph Layout customization integration sample

```
myVisualizer.balloonLayout.nodesSpacing=15;
myVisualizer.balloonLayout.useEvenAngles=false;
myVisualizer.balloonLayout.rootSelectionPolicy=LayoutConstants.DIRECTED_ROOT_SELECTION;
myVisualizer.hierarchicalTreeLayout.layerDistance=40;
myVisualizer.hierarchicalTreeLayout.defaultNodeDistance=40;
myVisualizer.hierarchicalTreeLayout.orientation=LayoutConstants.ORIENTATION_LEFT_RIGHT;
browser.hierarchicalTreeLayout.edgeDrawing=LayoutConstants.ORTHOGONAL_STRAIGHT_POLYLINE;
myVisualizer.radialLayout.minimalRadius=100;
browser.radialLayout.edgeDrawing=LayoutConstants.ORTHOGONAL_CURVED_POLYLINE;
browser.radialLayout.rootSelectionPolicy=LayoutConstants.DIRECTED_ROOT_SELECTION;
```

Expand, Collapse and Visibility Level

Visualizer can automatically detect hierarchies from any data input and layout them, however in some cases, hierarchies are too large to be easily understood. Visualizer gives solutions to a better user experience with navigation via:

- Visibility Level (**visibilityLevel**) : defines the last level allowed to be shown, thus Visualizer user will have the choice of his browsing path according to his preferences.

- Expand/Collapse Button: This button is shown by default in Visualizer component and enables expand and collapse control on a given hierarchy sub-Tree. This Button can be hiden by setting the **showExpandCollapseButton** to false.
- Expand/Collapse functionality on Double Click: This option is disabled by default as the Expand/Collapse button is show by default, but can be enabled by setting the **expandOnDoubleClick** property to true.
  The Expand/Collapse Functionality merged to Visibility level, bring better hierarchy navigation experience and simpler way for viewing Hierarchies. All these features can be controlled used properties described previously.

Related Visualizer Properties: ***visibilityLevel, showExpandCollapseButton, expandOnDoubleClick.***

Events

Visualizer shares its status and user interaction facts using events. Any developer need just to register (listen) to one of these events to perform a given task.
The events are listed in the following table:

VisualizerEvent property	Description
elementClicked	Indicates that an element has been clicked.
elementDoubleClicked	Indicates that an element has been double clicked.
elementExpanded	Indicates that an element has been expanded.
elementCollapsed	Indicates that an element has been collapsed.
elementsStatusChanged	Indicates All elements which Expand/Collapse Status have been changed.
elementRollOver	Indicates that an element is being rolled over.
elementRollOut	Indicates that an element is being rolled out.
elementsDragStarted	Indicates that elements started to be dragged by the user.
elementsDragFinished	Indicates that the user stopped dragging elements.
scrollBegin	Indicates that the user has started scrolling inside Visualizer.
scrollEnd	Indicates that the user has ended a scrolling operation inside Visualizer.
visibilityLevelChanged	Indicates that the visibility level has been changed. The event contains first level <code>GenericSprite</code> elements that have been collapsed or expanded.
animationStopped	Indicates that Visualizer content animation has ended.
dataLoaded	Indicates that Visualizer data input has been loaded and rendered.

.