Visualization of Context Graphs - JUNG and Zest

Nihal ABLACHIM

Supervisor: Ş.l. dr. ing. Andrei Olaru
University "Politehnica" of Bucharest

February 2013

Table of contents

1. Introduction
2. Java Universal Network/Graph Framework(JUNG)
   - What is JUNG?
   - More about JUNG
   - How do we create graphs in JUNG?
   - How do we visualize graphs in JUNG?
   - What kind of algorithms does JUNG provide?
3. Zest
   - What is Zest?
   - How do we create graphs in Zest?
   - How do we visualize graphs in Zest?
   - What kind of algorithms does Zest provide?
4. Conclusions
5. Future Work
6. References

Introduction

- The purpose of the main research is to develop an application that allows the user to edit his context graph and that automatically detects the situation of the user and proposes appropriate action, based on pre-existing context graphs.

Introduction

- The purpose of the main research is to develop an application that allows the user to edit his context graph and that automatically detects the situation of the user and proposes appropriate action, based on pre-existing context graphs.
- In this circumstances, a starting point could be to implement a graphical interface which permits the user to visualize and dinamically edit his context graphs. Since there are already free and open-source softwares that provide the manipulation and visualization of the graphs there is no need to reinvent the wheel and implement another framework but to make use of what already exists.

What is JUNG?

What is JUNG?

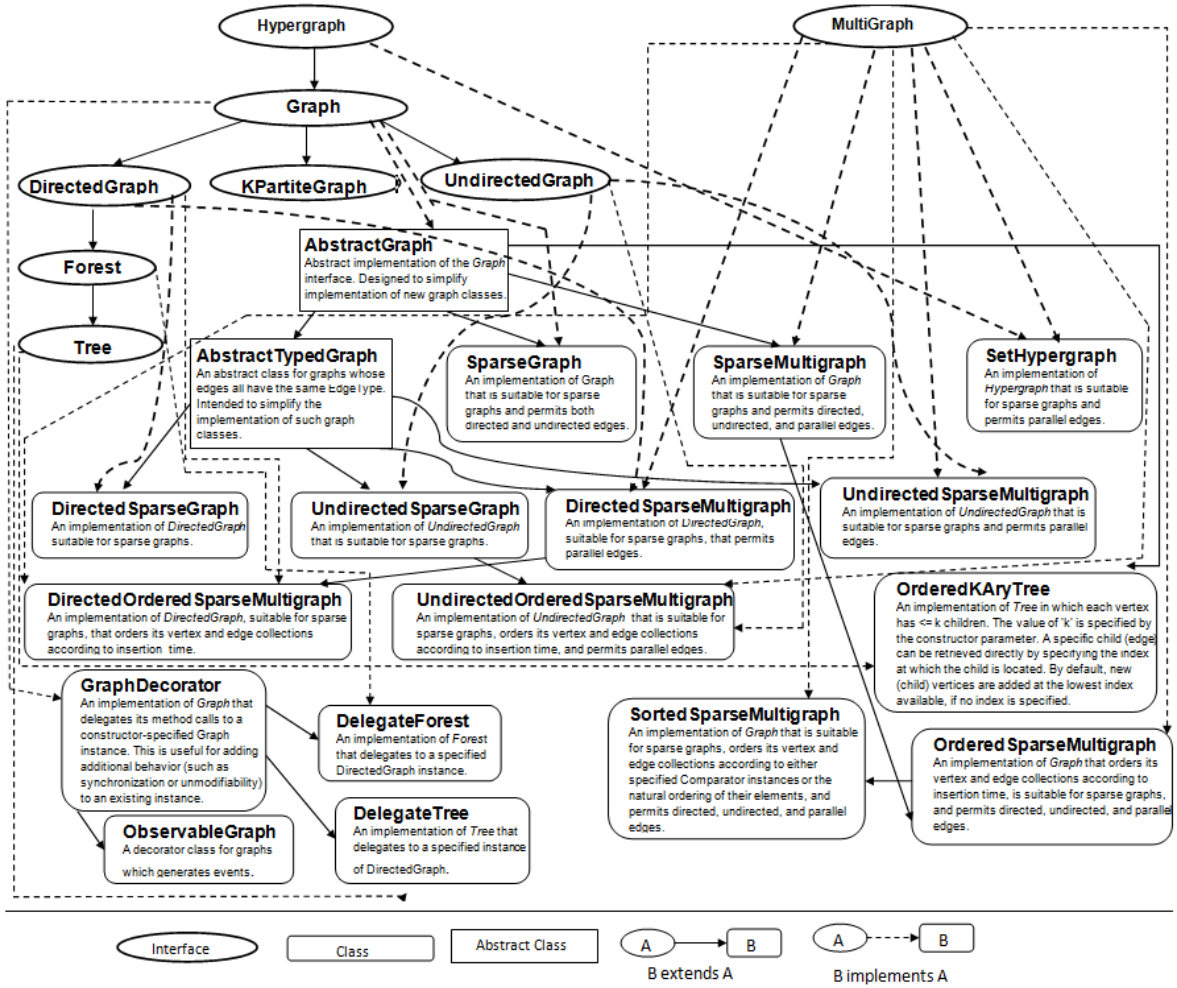
Framework for the modeling, analysis, and visualization of graphs in Java.

- supports most types of graphs
- separate, flexible visualization framework
- "rich" library of algorithms

More about JUNG

- Open-source software
- Written in Java
- Created by 3 UCI CS PhD students:
  - Scott White
  - Joshua OMadadhain
  - Danyel Fisher

Graph Types


Hypergraph
MultiGraph
Graph
DirectedGraph
KPartiteGraph
UndirectedGraph
Forest
Tree
AbstractGraph
Abstract implementation of the Graph interface. Designed to simplify implementation of new graph classes.
AbstractTypedGraph
An abstract class for graphs whose edges all have the same Edge Type. Intended to simplify the implementation of such graph classes.
SparseGraph
An implementation of Graph that is suitable for sparse graphs and permits both directed and undirected edges.
SparseMultigraph
An implementation of Graph that is suitable for sparse graphs and permits directed, undirected, and parallel edges.
SetHypergraph
An implementation of Hypergraph that is suitable for sparse graphs and permits parallel edges.
Directed SparseGraph
An implementation of DirectedGraph suitable for sparse graphs.
Undirected SparseGraph
An implementation of UndirectedGraph that is suitable for sparse graphs.
Directed SparseMultigraph
An implementation of DirectedGraph, suitable for sparse graphs, that permits parallel edges.
Undirected SparseMultigraph
An implementation of UndirectedGraph that is suitable for sparse graphs and permits parallel edges.
DirectedOrdered SparseMultigraph
An implementation of DirectedGraph, suitable for sparse graphs, that orders its vertex and edge collections according to insertion time.
UndirectedOrdered SparseMultigraph
An implementation of UndirectedGraph that is suitable for sparse graphs, orders its vertex and edge collections according to insertion time, and permits parallel edges.
OrderedKAryTree
An implementation of Tree in which each vertex has <= k children. The value of 'k' is specified by the constructor parameter. A specific child (edge) can be retrieved directly by specifying the index at which the child is located. By default, new (child) vertices are added at the lowest index available, if no index is specified.
GraphDecorator
An implementation of Graph that delegates its method calls to a constructor-specified Graph instance. This is useful for adding additional behavior (such as synchronization or unmodifiability) to an existing instance.
DelegateForest
An implementation of Forest that delegates to a specified DirectedGraph instance.
Sorted SparseMultigraph
An implementation of Graph that is suitable for sparse graphs, orders its vertex and edge collections according to either specified Comparator instances or the natural ordering of their elements, and permits directed, undirected, and parallel edges.
Ordered SparseMultigraph
An implementation of Graph that orders its vertex and edge collections according to insertion time, is suitable for sparse graphs, and permits directed, undirected, and parallel edges.
ObservableGraph
A decorator class for graphs which generates events.
DelegateTree
An implementation of Tree that delegates to a specified instance of DirectedGraph.
Interface
Class
Abstract Class
A
B
B extends A
A
B
B implements A

Creating a graph in JUNG

- The simplest way to create a graph is by calling the constructor for the desired type of graph

```
Graph<Integer, String> g =new SparseMultigraph<Integer, String>();
```

- Adding vertices and edges

```
g.addVertex((Integer) 1);
g.addVertex((Integer) 2);
g.addVertex((Integer) 3);
g.addEdge("Edge-A", 1, 2, EdgeType.DIRECTED);
g.addEdge("Edge-B", 2, 3);
```

- Removing vertices and edges

```
g.removeVertex(1);
g.removeEdge("Edge-A");
```

What do we need to visualize graphs in JUNG?

- A Graph to be visualized.

What do we need to visualize graphs in JUNG?

- A Graph to be visualized.
- A Layout, which takes the graph and determines the location at which each of its vertices will be drawn. JUNG provides many different layout algorithms for positioning the vertices of a graph(e.g. CircleLayout, RadialTreeLayout, SpringLayout, TreeLayout etc.).

What do we need to visualize graphs in JUNG?

- A Graph to be visualized.
- A Layout, which takes the graph and determines the location at which each of its vertices will be drawn. JUNG provides many different layout algorithms for positioning the vertices of a graph(e.g. CircleLayout, RadialTreeLayout, SpringLayout, TreeLayout etc.).
- A (Swing) Component, which provides a drawing area upon which the data is rendered. The basic class for viewing graphs in JUNG is the BasicVisualizationServer class(*edu.uci.ics.jung.visualization*). This implements the JUNG *VisualizationServer* $< V, E >$ interface and inherits from Swing's JPanel class (javax.swing.JPanel).

What do we need to visualize graphs in JUNG?

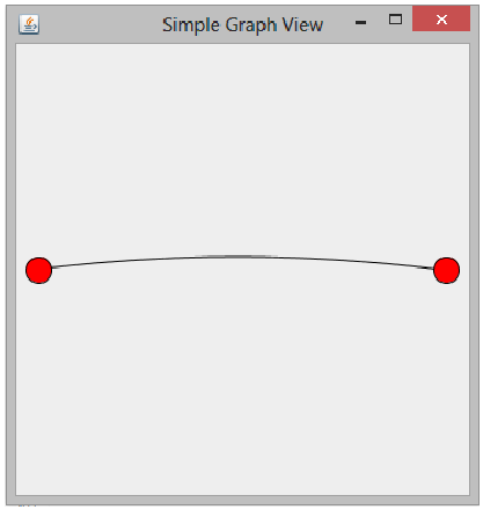
- A Graph to be visualized.
- A Layout, which takes the graph and determines the location at which each of its vertices will be drawn. JUNG provides many different layout algorithms for positioning the vertices of a graph(e.g. CircleLayout, RadialTreeLayout, SpringLayout, TreeLayout etc.).
- A (Swing) Component, which provides a drawing area upon which the data is rendered. The basic class for viewing graphs in JUNG is the BasicVisualizationServer class(*edu.uci.ics.jung.visualization*). This implements the JUNG *VisualizationServer* $< V, E >$ interface and inherits from Swing's JPanel class (javax.swing.JPanel).
- A Renderer, which takes the data provided by the Layout and paints the vertices and edges into the provided Component.

Simple Graph Display

```
Layout<Integer, String> layout = new CircleLayout<Integer,String>(g);
layout.setSize(new Dimension(350, 350));
 // sets the initial size of the space
BasicVisualizationServer<Integer, String> vv = new BasicVisualizationServer<Integer, String>(layout);
vv.setPreferredSize(new Dimension(350, 350));
// Sets the viewing area size
JFrame frame = new JFrame("Simple Graph View");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(vv);
frame.pack();
frame.setVisible(true);
```

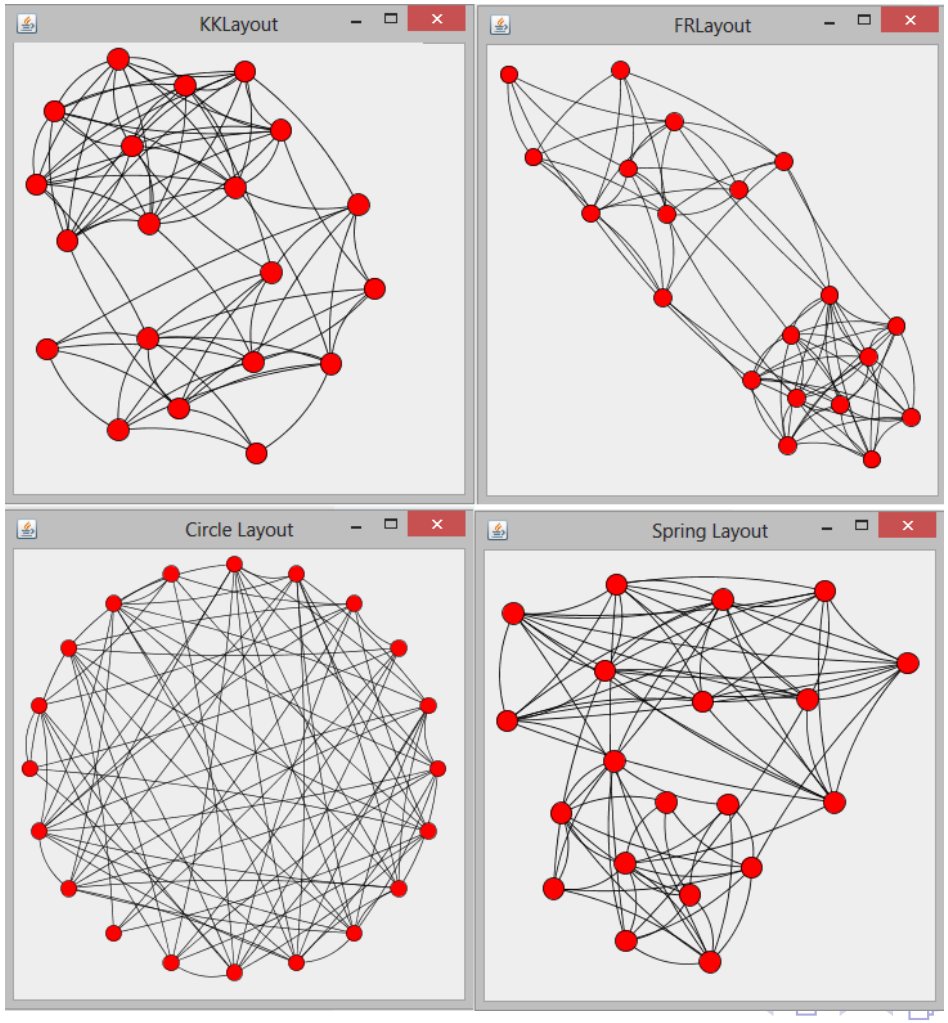
Simple Graph Display

```
Layout<Integer, String> layout = new CircleLayout<Integer,String>(g);
layout.setSize(new Dimension(350, 350));
 // sets the initial size of the space
BasicVisualizationServer<Integer, String> vv = new BasicVisualizationServer<Integer, String>(layout);
vv.setPreferredSize(new Dimension(350, 350));
// Sets the viewing area size
JFrame frame = new JFrame("Simple Graph View");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(vv);
frame.pack();
frame.setVisible(true);
```



Graphs displayed with different kind of layout algorithms

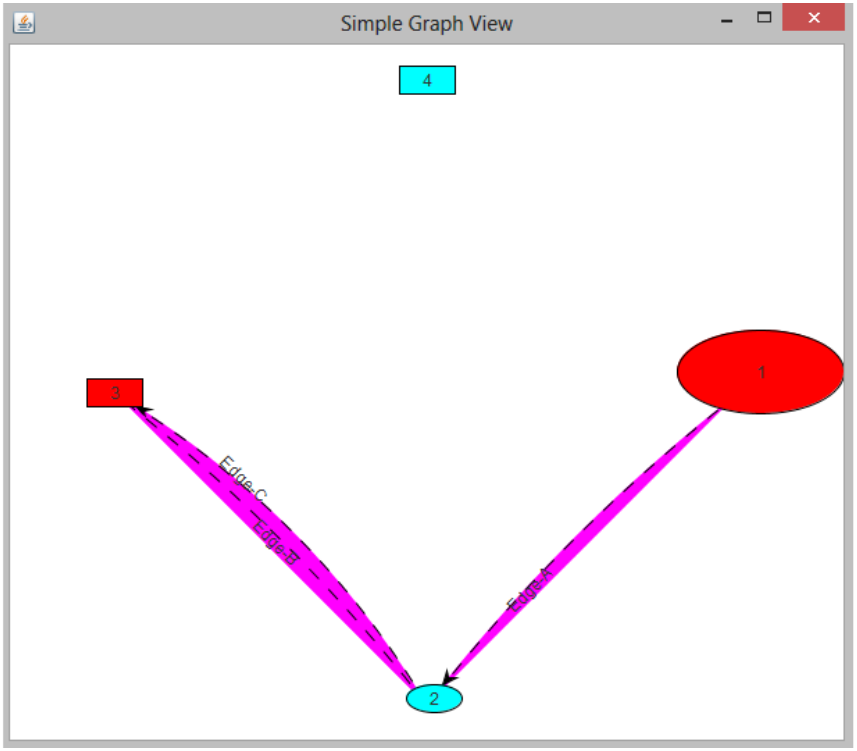
Graphs displayed with different kind of layout algorithms



Graph view customization

The default implementation fetches the location of each vertex from the Layout, paints each one with the Renderer inside the Swing Component, and paints each edge as a straight line between its vertices. Users may customize this behavior as desired; JUNG includes utilities and support classes that facilitate such customization.

Customized view of a graph in JUNG



What kind of algorithms does JUNG provide?

- Layout Algorithms
  - FRLayout(Fruchterman-Rheingold)
  - KKLayout(Kamada-Kawaii)
  - RadialLayout
  - TreeLayout
  - RadialTreeLayout
  - CircleLayout
  - SpringLayout
- Paths problem solving algorithms
  - DijkstraShortestPath
  - MinimumSpanningForest
  - UnweightedShortestPath
- Clustering algorithms
- Transformation algorithms

What is Zest?

What is Zest?

Zest is a set of visualization graphs/networks built for Eclipse.

- It has been developed in SWT/Draw2D.
- Open-source software
- Written in Java

Creating a graph in Zest

- The simplest way to create a graph is by declaring the graph as a *Graph* type and then calling the constructor for *Graph*:

```
graph = new Graph(parent, SWT.NONE);
```

- Declaring the vertices

```
GraphNode node1 = new GraphNode(graph, SWT.NONE, "1");
GraphNode node2 = new GraphNode(graph, SWT.NONE, "2");
GraphNode node3 = new GraphNode(graph, SWT.NONE, "3");
GraphNode node4 = new GraphNode(graph, SWT.NONE, "4");
GraphNode node5 = new GraphNode(graph, SWT.NONE, "5");
GraphNode node6 = new GraphNode(graph, SWT.NONE, "6");
```

Creating a graph in Zest - cont.

- Declaring the edges

```
GraphConnection c1=new GraphConnection(graph, ZestStyles.NONE, node1,node2);
c1.setText("2");
GraphConnection c2=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node1,node6);
c2.setText("1");
GraphConnection c3=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node1,node4);
c3.setText("4");
GraphConnection c4=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node2,node3);
c4.setText("1");
GraphConnection c5=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node2,node5);
c5.setText("1");
GraphConnection c6=new GraphConnection(graph, ZestStyles.CONNECTIONS_DIRECTED, node4,node3);
c6.setText("2");
```

- Removing vertices and edges

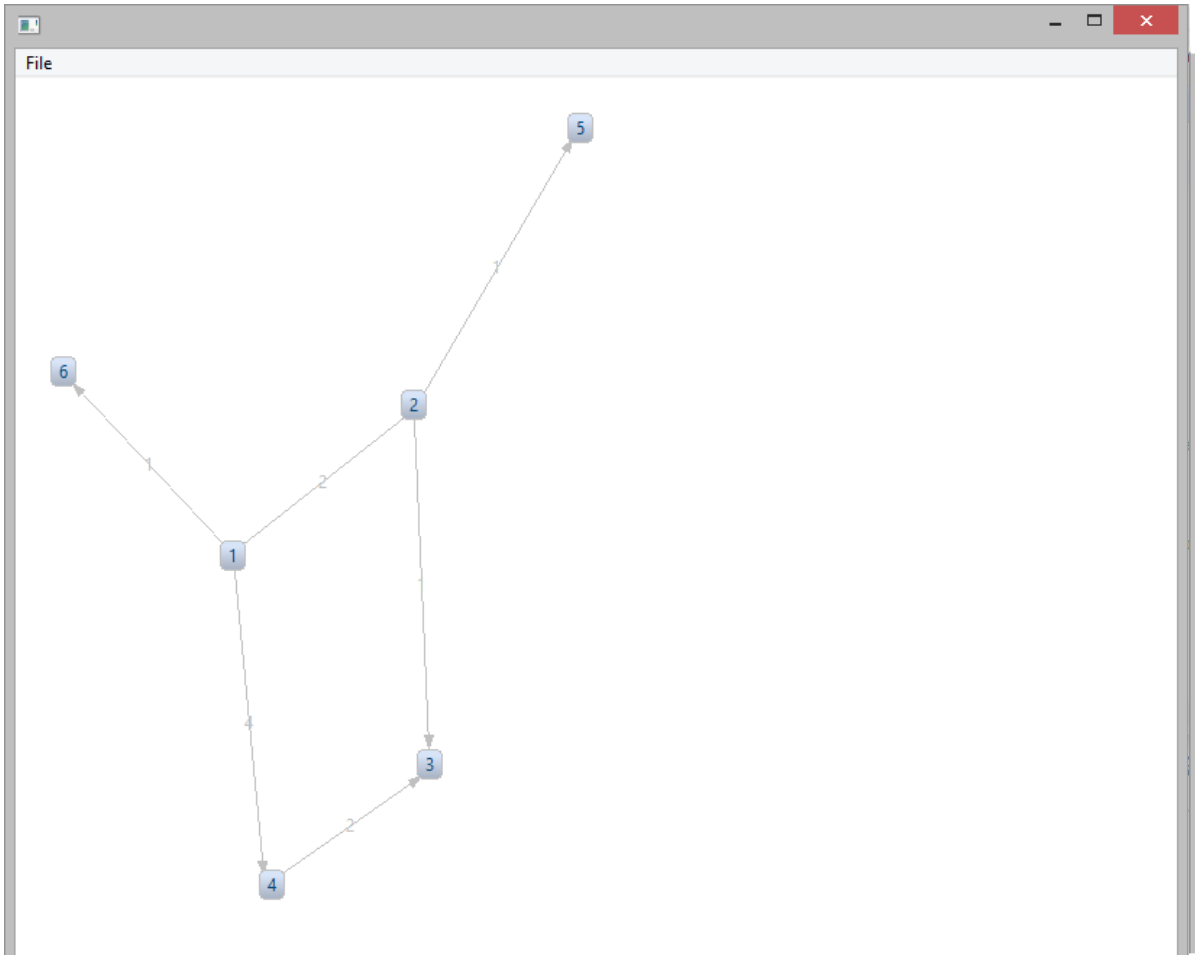
```
node3.dispose();
c3.dispose();
```

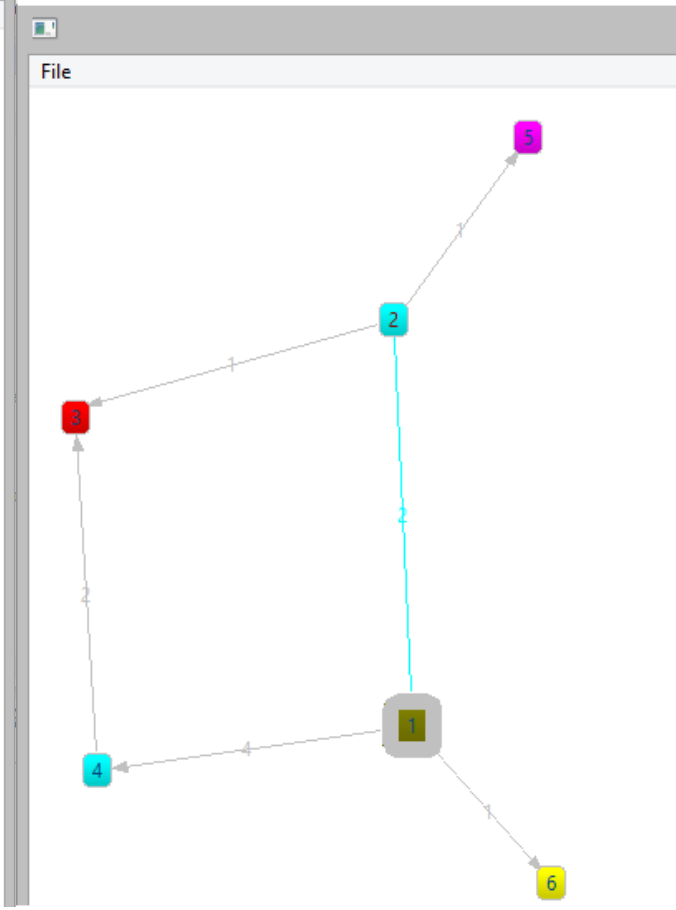
How do we visualize graphs in Zest?

Eclipse Zest provides graph layout managers. A graph layout manager determines how the nodes (and the edges) of a graph are arranged on the screen.
The simplest way to visualize a graph in Zest is by declaring an instance of one of the layouts provided by Zest and then calling the method setLayoutAlgorithm() of the graph:

```
SpringLayoutAlgorithm SLA= new SpringLayoutAlgorithm(LayoutStyles.NO_LAYOUT_NODE_RESIZING);
graph.setLayoutAlgorithm(SLA, true);
```

Graph visualization in Zest





What kind of algorithms does Zest provide?

Unlike JUNG which has a large variety of algorithms, Zest has only layout managing algorithms:

- SpringLAyoutAlgorithm
- HorizontalLayoutAlgorithm
- GridLayoutAlgorithm
- TreeLayoutAlgorithm
- RadialLayoutAlgorithm

Conclusions

- Both are toolkits for creating and visualizing graphs.
- JUNG provides more algorithms then Zest.
- JUNG is more documented then Zest.
- Zest has more recent releases(Zest 3.8.0-2012) then JUNG(JUNG2-2010).

Future Work

- Choose one of the tools presented
- Implement a graphical user interface(GUI) which permits the visualization and interactively editing of context graphs.

References

[1] Andrei Olaru and Adina Magda Florea and Amal El Fallah Seghrouchni, *Graphs and Patterns for context awareness*, 2011.

[2] http://jung.sourceforge.net/

[3] http://stackoverflow.com/

[4] http://www.vogella.com/articles/EclipseZest/article.html