Graph Layout Algorithms and Libraries: Overview and Improvements

Renata Vaderna, Gordana Milosavljević, Igor Dejanović

Faulty of Technical Sciences, University of Novi Sad, Serbia
{vrenata, grist, igord}@uns.ac.rs

***Abstract*—This paper focuses on exploring the possibilities of applying graph drawing algorithms to lay out custom diagrams, with emphasis put on UML class diagrams. Implementing even the simplest of layout algorithms that would lead to acceptable results requires excessive knowledge of graph theory. For this reason, many developers have to rely on existing solutions. There are several open source Java libraries for graph drawing and analysis, but most of them come with certain problems and limitations making their integration with a separately developed graphical editor overly complex. To deal with those issues, we are developing another graph drawing and analysis library, called Grad.**

I. Introduction and Motivation

When developing modeling tools or expanding existing ones, a need to automatically lay out diagram elements in an aesthetically pleasing way might arise.

One such example is a lightweight UML class diagram editor implemented as a part of a larger tool called Kroki [1, 14]. Kroki enables users to create sketches of business applications using several embedded tools, thus enabling each participant to use their preferred way of development: mockup editor, command console, or the mentioned lightweight UML class diagram editor. On top of that, sketches can be imported from general purpose modeling tools. Furthermore, the class diagram editor should be capable of opening imported sketches and sketches created using other Kroki tools i.e. showing class diagrams which represent them. Changes made in Kroki's class diagram editor are immediately visible in the mockup editor and vice versa, which explains the need for automatically arranging newly created elements (packages, classes and links between them).

Implementing even the simplest of layout algorithms that would lead to acceptable results requires excessive knowledge of graph theory. Furthermore, simply deciding which class of graph drawing algorithms would be best suited for the given application can be challenging for those new to this area of mathematics. For these reasons, many developers would have to rely on existing solutions. There are many open source libraries which focus on graph drawing and provide implementation of certain layout algorithms. With a large number of excellent graph libraries for C/C++ and Python, it should be emphasized that only Java libraries will be considered in this paper, such as the popular JGraphX, JGraphT, JUNG and Prefuse.

Although providing a decent number of layout algorithms, all of these libraries primarily focus on graph visualization, thus making simply calling the desired algorithm and retrieving the results overly complex. In addition to this, it is very unlikely that all elements of a certain diagram would be of the same size, which would require usage of a layout algorithm which takes this into consideration. Some of the available solutions, however, do not. On top of that, many algorithms handle recursive links and multiple links between the same two elements quite poorly. However, such links frequently appear in class diagrams, so these problems cannot be ignored.

Having the mentioned limitations in mind, we are developing another open source graph drawing and analysis library, called Grad (GRaph Analysis and Drawing) [2]. Unlike the other ones, it puts a lot of emphasis on the ease of integration with other graphical editors and deals with the previously mentioned problems.

The rest of the paper is structured as follows. Section 2 explains the need for automatically applying layout algorithms within an existing graphical editor by shortly describing Kroki's lightweight UML editor and the requirements it had to fulfill. Section 3 gives a brief introduction to graph theory and graph drawing algorithms. Section 4 showcases some popular Java graph drawing and analysis libraries and points out some of the problems encountered when integrating them with separately developed editors. Section 5 presents Grad, a library being developed in order to address the most common integration issues. Finally, section 6 concludes the paper and outlines future work.

II. Kroki's Lightweight UML Editor

A lightweight UML class diagram editor was developed as a part of a tool named Kroki. Kroki is used for rapid prototyping and participatory development of enterprise applications based on mockups. It enables users to create sketches of applications using a mockup editor, command console, by importing models from general purpose modeling tools and by using the mentioned UML class diagram editor. Changes made in Kroki's class diagram editor should immediately be visible in the mockup editor and vice versa. Therefore, Kroki's UML class diagram editor had to fulfill some additional requirements.

Firstly, packages, classes and their attributes and methods and links established between them should have additional semantics as they need to represent certain elements of the sketches. This leads to the conclusion that simply being able to visualize the sketches as class diagrams isn't enough.

Secondly, it should be possible to open diagrams corresponding to sketches created using other Kroki tools than the UML editor with it. When doing so, there is no

data available regarding positions of the UML classes formed from certain elements of the sketch. Therefore, a layout algorithm must be automatically performed. Without that, users would have to lay out the diagrams manually. Since these diagrams can be quite large, placing all of the elements in the desired positions would drastically slow down the use of the Kroki tool.

III. Basic Graph Theory Concepts and Graph Drawing Algorithms

In the following section, a short introduction to graph drawing theory, as well as an overview of the most commonly used algorithms will be given.

A. Basic definitions

A graph (V, E) is an ordered pair consisting of a finite set V of vertices and a finite set E of edges, that is, pairs (u,v) of vertices. A path is a sequence of distinct vertices, $v_1, v_2, \ldots, v_k$, with $k \geq 2$, together with the edges $(v_1, v_2), \ldots, (v_{k-1}, v_k)$. A cycle is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$, with $k \geq 2$, together with the edges $(v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_1)$ [3] .

If edges are unordered pairs of vertices, then the graph is *undirected*. On the other hand, if edges are ordered pairs of vertices, the graph is *directed*. A graph is said to be *connected* if there is a path from any vertex to any other vertex in the graph. Graphs which are not connected are referred to as *disconnected*. Graphs which contain at least one cycle are called *cyclic* graphs, while the ones that do not are known as *acyclic*. A graph is *simple* if it doesn't contain any edges that join a vertex to itself (loops) or more than one edge connecting the same two vertices (multiple edges). Graphs which permit multiple edges are called *multigraphs*.

A drawing Γ of a graph G maps each vertex v to a distinct point Γ(v) of the plane and each edge (u, v) to a simple open Jordan curve Γ(u, v) with endpoints Γ(u) and Γ(v) [3]. A drawing is *planar* if no two distinct edges intersect except, possibly, at common endpoints. Some algorithms for constructing drawings of graphs are only designed for special classes of graphs, like trees, (simple, undirected, connected acyclic graphs), planar graphs (graphs which can be drawn in a plane without edges crossing), or directed acyclic graphs, while the other ones even work for general graphs.

B. An overview of graph drawing algorithms

An overview of the most popular classes of graph drawing algorithms will be given in the next couple of paragraphs. More detailed descriptions of them can be found in [3].

Tree drawing is one of the best studied areas of graph drawing. That is not surprising since automatic generation of drawings of trees finds many practical applications. All trees are planar, which means that it is always possible to construct drawings of them with no edge crossings. There are several time-efficient tree drawing strategies which allow creation of aesthetically pleasing drawings.

A circular drawing of a graph is its visualization with the following characteristics:

- the graph is partitioned into clusters
- the nodes of each cluster are placed onto the circumference of an embedding circle
- each edge is drawn as a straight line

These algorithms have many application, especially in tools that manipulate networks.

A rectangular drawing of a plane graph is a drawing of it in which each vertex is drawn as a grid point on an integer grid and each edge is drawn as a sequence of alternate horizontal and vertical line segments along the grid. These algorithms find applications in circuit layouts, database and entity-relationship diagrams and floorplanning.

Force-directed algorithms are among the most important classes of graph drawing algorithms. They are very flexible and can be used to calculate layouts of all simple undirected graphs. They calculate the layout of the graph using only information contained within the structure of the graph itself. Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries, and tend to produce crossing-free layouts for planar graphs. There are many force-driven algorithms, with Tutte's 1963 barycentric method being considered to be the first one. The most popular ones include Kamada and Kawai [4] and Fruchterman-Reingold [5].

Hierarchical drawing algorithms can be used when dealing with directed graphs (or *digraphs*) which represent hierarchies. Examples of hierarchies or near-hierarchies are, among others, class diagrams and function call graphs from software engineering. The main idea behind hierarchical methods is to modify force-directed methods to take into account edge directions, and use them to draw digraphs.

C. Class diagrams as graphs

Class diagrams can easily be viewed as graphs with the elements representing vertices and the links representing edges. They can have multiple links between the same two elements. On top of that, these diagrams can contain recursive links (links connecting one element to itself). Therefore, they can be viewed as multigraphs that can contain loops. These graphs can be planar, but there is no guarantee that that will be the case. The same goes for connectivity. Similarly, some class diagrams contain cycles, others are acyclic. Generally, class diagrams are directed, but this cannot be seen as a rule since links can, and often are navigable, but not always.

Having all of this in mind, as well as the descriptions of different types of graph drawing algorithms, it can be concluded that force-directed and hierarchical algorithms are most suitable for usage in class diagram layouters. However, simply performing these algorithms might not be enough to form an aesthetically pleasing drawing of a class diagram. Additional steps might have to be performed in order to show loops and multiple edges correctly.

IV. Related Work

There are quite a few libraries for analyzing and drawing graphs for Java. In this section, some of the most popular ones will be briefly described, with the focus being on quantity and quality of the implemented graph layout algorithms. Furthermore, a few examples of how these algorithms can be used by some other projects will be given, accompanied by a short discussion regarding the complexity of such calls.

A. A preview of the most widely used free graph libraries for Java

The most widely known and used free graph libraries include JGraphT [6] and JGraph [7], JGraphX [7], Prefuse [8] and JUNG (Java Universal Network/Graph Framework) [9]. It can also be noted that there are some commercial solutions, such as yFiles from yWorks [10], but, since using them in most projects is not a likely possibility, they will not be considered in this paper. Furthermore, neither will tools that only generate images, such as GraphViz [11], since their layout algorithms cannot be integrated with already existing graphical editors.

All of these libraries focus on enabling users to model and analyze and/or visualize data that can be represented as a graph or a network. Apart from JGraphT, all projects put heavy emphasis on visualization, some even allowing users to interact with the created graphs. In the following passages a short preview of some of the most popular Java graph libraries will be given, followed by a discussion regarding how convenient or inconvenient it would be to integrate their layout algorithms with already existing tools for visualizing the given data.

JGraphT is a free Java graph library that provides mathematical graph-theory objects and algorithms [6]. It enables simple graph creation and offers implementations of a wide range of graph analysis algorithms, such as Dijkstra's shortest path, but does not provide any layout algorithms. In fact, it relies on **JGraph** for visualization. A notable problem which users of this particular combination of libraries face is that JGraphT only supports usage of an older version of JGraph. JGraph was significantly enhanced and rewritten from scratch in version 6, with even the name being changed to JGraphX [7]. However, JGraphT wasn't updated, still using the old version of the previously mentioned visualization library.

JGraphX is a Java Swing graph visualization library. It enables integration of interactive diagrams into larger Swing applications [7]. It is possible to customize certain properties of the graphs such as design of the vertices, labels of the edges, etc. Most importantly, it also provides a few layout algorithms meant to assist users in setting out their graph. Most notably, JGraphX implements one rather effective force-directed algorithm, a simulated annealing layout based on [12]. Moreover, it also provides implementations of a few different tree layouts.

Prefuse is another library set of tools for creating interactive data visualizations [8]. Its distinguishing feature is the ability to read data and create graphs directly from XML files and relational databases with only a line or two of code. When it comes to layout algorithms, Prefuse, like JGraphX offers a number of tree layouts, but also two force-directed ones, including the mentioned Fruchterman-Reingold.

Java Universal Network/Graph Framework, also known as JUNG, is a library that offers both the possibility of analyzing and visualizing graphs [9]. The current distribution of JUNG includes implementations of a number of algorithms from graph theory, data mining and social network analysis, but also provides a visualization framework. JUNG framework, while not containing the largest number of implementations of different layout algorithms out of the other mentioned alternatives, does implement more force-directed ones. In fact, JUNG provides an implementation of the previously mentioned Fruchterman-Reingold algorithm and a slightly modified version of it, as well as Kamada-Kawai. However, it is quite complex to set custom sizes of the JUNG graph vertices.

B. Integration with existing graphical editors

The problem which will be analyzed in this section is how to use layout algorithms provided by the mentioned libraries within an already existing graphical editor. To be more precise, within an existing class diagram editor, where sizes of the vertices play a significant role. With all of the graph drawing libraries putting strong emphasis on visualization, simply calling a layout algorithm and retrieving the results i.e. positions of the vertices and, if available, information about locations and shapes of the edges, can be quite complex.

Typically, in order to call a layout algorithm, it is necessary to provide an instance of the graph class, meaning that the application's data model has to be transformed into the suitable format. In addition to that, visualization components may have to be initialized, even though they won't be used. More importantly, implementations of layout algorithms and/or graph, vertex and edge classes have to be analyzed in search of a way of retrieving information about the vertices and edges following the execution of layout algorithms. Out of the mentioned libraries, JGraphX and JUNG provide the largest number and the most complex layout algorithms. For this reason, examples will cover integration with their algorithms.

It is worth mentioning that integration with Prefuse is even more complex. Prefuse enables simple creation of graphs directly from XML files and relational databases. While it is easy to see why these features could be put to good use in many projects, it is dynamical creation of graphs which is of importance in this particular case. That, however, is accomplished much harder. JGraphT - JGraph combination also won't be used in the examples, since it is now obsolete, like it was explained in the previous section.

Every diagram of Kroki's UML class editor contains a list of elements and links between them. Let's assume that prior to calling the layout algorithms, elements were already loaded into a list called *diagramElements*, while the links were all inserted into a list simply called *links*. An example of calling a JGraphX layout algorithm and retrieving positions of the vertices is shown in code listing 4.1.

Firstly, it can be noticed that creating graphs using already existing elements is a bit inconvenient as JGraphX graphs aren't parametrized and thus cannot contain vertices of any given class. Secondly, one must be quite familiar with how JGraphX works in order to get positions of the vertices once layout algorithms have finished calculating them.

Accomplishing the same using the JUNG framework is much simpler, which can be seen by analyzing code listing 4.2. However, it is necessary to initialize the visualization component in order to trigger execution of layout algorithms. Also, retrieving positions from layout algorithm after it was performed, while easy to do, is not well documented and can prove to be quite hard to discover.

```
mxGraph graph = new mxGraph();
graph.getModel().beginUpdate();
Object parent = graph.getDefaultParent();
Map<GraphElement, Object> elementsJGraphXVerticesMap =
        new HashMap<GraphElement, Object>();
try
{
    for (GraphElement element : diagramElements){
        Object jgraphxVertex = graph.insertVertex(parent, null,
                element, 0, 0, element.getSize().getWidth(),
                element.getSize().getHeight());
        elementsJGraphXVerticesMap.put(element, jgraphxVertex);
    }
    for (Link link : links){
        Object v1 = elementsJGraphXVerticesMap.get(link.getOrigin());
        Object v2 = elementsJGraphXVerticesMap.get(link.getDestination());
        graph.insertEdge(parent, null, null, v1, v2);
    }

}
finally{
    graph.getModel().endUpdate();
}
mxOrganicLayout jgraphxOrganic = new mxOrganicLayout(graph);
jgraphxOrganic.execute(parent);
for (Object vertex : elementsJGraphXVerticesMap.values()){
    mxIGraphModel model = graph.getModel();
    mxGeometry geometry = model.getGeometry(vertex);
    //finally, we can get the x and y coordiates
    System.out.println(geometry.getX() + ", " +  geometry.getY());
}
```

Code listing 4.1 Calling a JGraphX layout algorithm and retrieving the results

```
UndirectedSparseGraph<GraphElement, Link> graph =
        new UndirectedSparseGraph<GraphElement, Link>();

for (GraphElement element : diagramElements)
    graph.addVertex(element);

for (Link link : links)
    graph.addEdge(link, link.getOrigin(), link.getDestination());

FRLayout<GraphElement, Link> layouter =
        new FRLayout<GraphElement, Link>(graph);

//triggers layouting
new DefaultVisualizationModel<GraphElement, Link>(layouter);

for (GraphElement element : diagramElements){
    Point2D p = layouter.transform(element);
    System.out.println(p);
}
```

Code listing 4.2 Calling a JUNG layout algorithm and retrieving the results

Another limitation of the JUNG framework, which was already briefly mentioned, is the complexity of setting custom sizes of the vertices. A solution to this problem proposed in [13] includes the use of aspects and requires considerable knowledge of the framework. On the other hand, simply using the default sizes of the graph vertices when performing layout algorithms can ultimately lead to their overlapping. Elements of class diagram, of course, fall into this category, which limits direct applicability of the JUNG framework.

Furthermore, it must be stressed that, as discussed in the third section, class diagrams are multigraphs that can contain loops. If that is indeed the case, in addition to calling the algorithms of the chosen library, users would have to handle loops and set positions of the overlapping edges (which happens when the graph has several edges between the same two vertices) themselves. In addition to the already mentioned problems, many algorithms do not perform particularly well if disconnected graphs are passed to them. There is often too much free space between the disjoint parts of such graphs.

V. GRAPH ANALYSIS AND DRAWING LIBRARY (GRAD)

The main motivation behind the project was to implement a variety of graph analysis and drawing algorithms and enable very simple integration with already existing graphical editors, which includes the possibility of calling the layout algorithms and retrieving the results very easily, while also being able to specify certain properties of the vertices, such as their sizes. It is worth mentioning that the current version of Grad also offers implementations of several graph analysis algorithms, such as planarity testing and graph traversal. However, they are not the main focus of this paper and will not be described in more detail. All examples of diagrams that will be shown in this section were created using Kroki's lightweight UML editor.

A. Layouting implementation

At the moment, there are five different layout algorithms available: three force-directed ones, one circular and the so-called box layout, which places elements in a table-like structure. Special attention was given to graphs with loops and multiple edges, enabling any custom class diagram to be arranged in an aesthetically satisfying way, with no edges overlapping. The problem that was mentioned in the previous section, regarding disconnected graphs was also addressed. Parts of these graphs are arranged separately and positioned in such way that they are neither too far apart from each other, nor too close. In the following passages, examples of class diagrams arranged using different layout algorithms will be shown.

The three currently provided **force-directed algorithms** are Kamada-Kawai, Fruchterman-Reingold and the basic spring algorithm. After performing these algorithms, additional steps are taken to make sure that no vertices are overlapping. If distances between any two vertices are smaller than the specified limit, their positions are adjusted. Users can set the limit themselves before calling layout algorithms. If they do not choose to do so, a predefined value is used. An example of an arranged class diagram with recursive links and two links between classes “Panel6” and “Panel2” using Kamada-Kawai algorithm is shown in Figure 5.1.

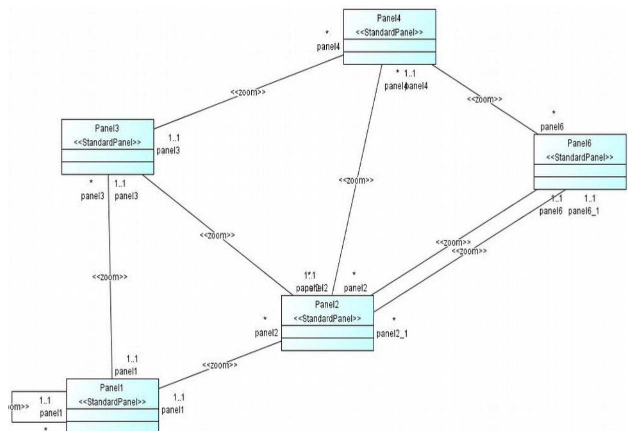

Figure 5.1 An example of a class diagram arranged using Kamada-Kawai algorithm

Like it was mentioned in section 3, force-directed algorithms tend to produce crossing-free layout for planar graphs. Looking at Figure 5.1, it can be noticed that this indeed is the case here. Also, the two links between the

same two classes don't overlap, and the recursive link connecting “Panel1” with itself is not hidden beneath the class.

Circular layout places vertices onto the circumference of an embedding circle. If the graph is biconnected (a graph which remains connected if any vertex is deleted), additional preprocessing is performed in order to minimize the number of crossings. The preprocessing involves calculation of the best possible order of vertices. An example of a class diagram arranged using the circular graph drawing algorithm is shown in Figure 5.2. Implementation of an algorithm for drawing non-biconnected graphs on multiple embedding circles is planned for future releases of Grad.

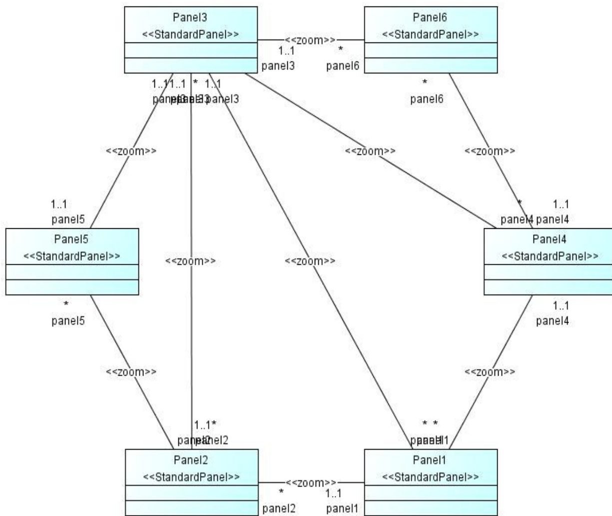

Figure 5.2 An example of a class diagram arranged using the circular graph drawing algorithm

Box layout places vertices in a table-like structure. The basic idea is to position a predefined number of vertices in one row, before continuing to the next one. Sizes of the vertices are taken into account when calculating heights of the rows and widths of the columns in order to prevent the vertices from overlapping. The number of vertices in a row can be adjusted by the user before executing the algorithm. If a class diagram is organized in such way that it contains a large number of packages on the first level, this layout is by far the best choice. An example of such usage is shown in Figure 5.3.

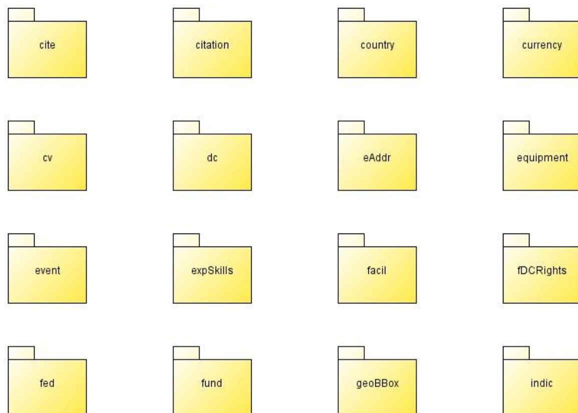

Figure 5.3 Using box layout to organize a diagram containing only packages

B. *Integration with existing graphical editors*

Grad can easily be used in combination with already existing graphical editors. In fact, the ease of integration was one of the project's main requirements.

The central class, which represents a graph, is parametrized, which means that it is safe to use just about any two classes as types of vertices and edges. The only requirement that must be fulfilled is that these classes have to implement appropriate interfaces (called *Vertex* and *Edge*), making it possible to easily specify properties of the vertices and edges (e.g. sizes of the vertices) which will later be used by the layout algorithms.

When calling a layout algorithm, one only needs to pass lists of vertices and edges, and not the already created graph since it will automatically be created later. Moreover, an object containing maps of vertices and edges and their positions is returned. Therefore, all information needed to position the editor's elements is obtained with no additional effort. Like it was already mentioned, Grad puts emphasis on properly handling loops and multiple edges. For this reason, edges can contain multiple segments whose endpoints are returned. A class diagram encapsulating previously explained is shown in Figure 5.4.

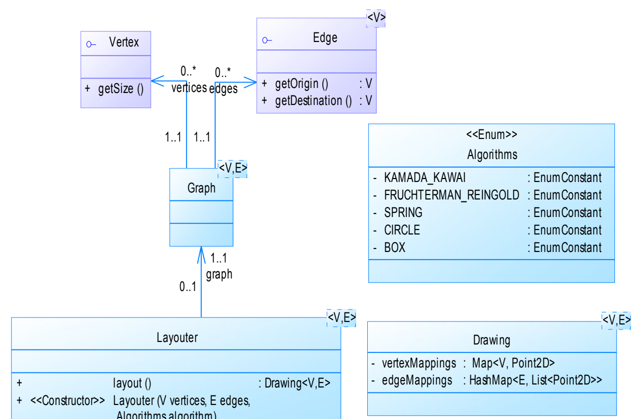

Figure 5.4 A part of Grad's model showing classes and interfaces needed for integration with other tools

It should be stressed that there is no need to dig deeper into Grad's implementation, to instantiate a visualization component or anything along those lines in order to call a layout algorithm and retrieve the results. To demonstrate that it is truly easy to do so, an example of calling the Kamada-Kawai graph drawing algorithm is shown in code listing 5.1. It is assumed that we have the same lists of elements and links like in chapter 4 at our disposal.

```
Layouter<GraphElement, Link> layouter =
        new Layouter<>(diagramElements,
                links, Algorithms.KAMADA_KAWAI);
Drawing<GraphElement, Link> drawing = layouter.layout();
Map<GraphElement, Point2D> elementPositions =
        drawing.getVertexMappings();
Map<Link, List<Point2D>> linkPosition =
        drawing.getEdgeMappings();
```

Code listing 5.1 Calling a Grad layout algorithm and retrieving the results

It can also be noted that it even isn't necessary to instantiate a class implementing the desired algorithm, as seen in code listing 5.1. The users only need to select an enumeration value which corresponds to their algorithm

of choice, while everything else will later be handled by the central class responsible for executing the layout algorithms, called *Layouter*.

VI. Conclusion

This paper presented an overview of different graph drawing algorithms and explored the possibility of integrating layout algorithms of some of the most popular Java graph drawing and analysis libraries with a separately developed graphical editor. In other words, in cases when their visualization isn't needed. Certain problems regarding ease of such integration, as well as some issues characteristic to arranging class diagrams were pointed out. They include the need to get quite familiar with the libraries before being able to call the layout algorithms they provide and retrieve the calculated positions of the vertices, as well as problems which might occur when the graph contains loops and multiple edges. These issues were addressed in a new graph drawing and analysis library called Grad.

Grad offers implementations of a variety of graph analysis and drawing algorithms, while focusing on the ease of integration with already existing graphical editors. Plans for future improvements of Grad include implementations of:

- an algorithm for drawing non-biconnected graphs on multiple embedding circles
- several tree drawing and hierarchical algorithms
- labeling algorithms which address automatic placement of text symbol labels

References

[1] G. Milosavljevic, M. Filipovic, V. Marsenic, D. Pejakovic, I. Dejanovic, “Kroki: A mockup-based tool for participatory development of business applications“, *IEEE 12th Conference on Intelligent Software Methodologies, Tools and Techniques*, pp. 235-242, 2013

[2] Graph Analysis and Drawing library (Grad), https://github.com/renatav/GraphDrawing, online, accessed January 11, 2015.

[3] Roberto Tamassia, *Handbook of Graph Drawing and Visualization*, Chapman & Hall/CRC, 2007.

[4] T. Kamada andS. Kawai, “An algorithm for drawing general undirected graphs”, in *Information Processing Letters,* vol. 31, pp. 7-15, April 1989.

[5] T. Fruchterman and E. Reingold, “Graph drawing by force-directed placement” in *Software Practice and Experience,* vol. 21, pp. 1129 – 1164, November 1991.

[6] JGraphT, http://jgrapht.org, online, accessed January 11, 2015.

[7] JGraphX, https://github.com/jgraph/jgraphx, online, accessed January 11, 2015.

[8] Prefuse, http://prefuse.org, online, accessed January 11, 2015.

[9] JUNG Framework, http://jung.sourceforge.net, online, accessed January 11, 2015.

[10] yFiles, www.yworks.com/en/products/yfiles/, online, accessed January 11, 2015.

[11] GraphViz, http://www.graphviz.org, online, accessed January 11, 2015.

[12] Ron Davidson, David Harel, “Drawing Graphs Nicely Using Simulated Annealing”, in *ACM Transaction on Graphics,* vol. 15, pp. 301-331, October 1996.

[13] Setting custom sizes of the JUNG graph vertices, http://sourceforge.net/p/jung/discussion/252062/thread/0b98adc5, online, accessed January 22, 2015.

[14] Kroki source, https://github.com/KROKIteam/KROKI-mockup-tool, online, accessed January 22, 2015.