JetUML diagram layout enhancements

Yingjie Xu *

April 2021

Contents

1 Introduction

All the UML drawing applications would require methods for handling edge drawings. Generally, there would be two approaches. The first one is to let the application draw all the edges based on some algorithm without giving the edit permission to users. Another approach would be letting the users edit the edge layout. The benefit of the second approach is quite obvious that users could edit the connection points and edge paths based on their needs.

*Supervised by professor Martin Robillard as a COMP 400 project during winter 2021 semester at McGill University.

However, for our specific needs in JetUML, we decided to move on with the first approach. Making edge layout editable is against the minimalist vision for JetUML[1].

JetUML is a lightweight application for drawing UML diagrams, the algorithm for calculating edge layout is an important part of the application. We want to have a design that would make the edge paths smarter and try to avoid as many edge crossings as possible. To avoid edge crossings, an edge priority scheme is designed to have a systematic way of calculating the edge layout. The class diagram is the most complicated diagram for having a clear edge layout and this report would cover some ideas about the updated algorithm.

During the exploration of updating the algorithm, we found that the original stateless design for calculating and drawing the edges prevents us from caching the paths for other edges. As a result, there are a lot of duplicated calculations. To solve this issue, we decided to add another layer named "layouter" for pre-calculating the edge paths before drawing the edges. By applying this idea, we could cache previous edge paths and reduce a lot of duplicated calculations.

2 Feature Description

The redesigned algorithm for calculating the layout would take node positions and other edge paths into consideration. By applying the priority scheme, we could make sure a consistent behaviour for drawing the edges. The higher priority edge would always be assigned a more centred position compared with the edge with a lower priority. By considering positions of start and end nodes, we would aggregate edges of the same category and on the same side which helps to avoid edge crossings.

The edge drawing procedure is also updated. Previously, we had stateless EdgeViewers which would not only calculate the edge path but also draw the edges. However, the stateless design of EdgeViewer triggers a lot of duplicated edge path calculation and makes it really hard to track other edge paths which would be useful for avoiding edge crossing. With the updated design, we separate the responsibility of calculating the edge path into another layer named "layouter". The layouter would first calculate the edge path for each edge and then the EdgeViewer would only need to draw the edge based on

[1]Cited from `https://github.com/prmr/JetUML/issues/375`

the pre-planned edge paths. The benefit of the layouter design is that it is stateful and opens the door to calculate new edge paths based on previously calculated edge paths. It also allows us to cache the path information for future queries.

3 Design Overview

This section gives an overview of the design [2]. It covers two parts, the first part is about the updated algorithm for class diagram edge layouting, the second part is about the updated design of layouter.

3.1 Edge layout algorithm for class diagram

The purpose of updating the edge layout algorithm is to make it smarter and avoid as many edge crossings as possible. In order to do this, we established the rules below.

Definition of position We first define the concept of position on a side of a node (figure:1). The midpoint of a node side would be the central point with an index of 0. Based on the central point, we increment the index separately on both the left and right sides. This concept would be used in the explanations below.

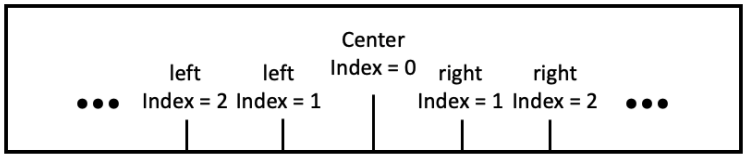

Figure 1: The concept of position

Edge category We categorize edges based on their visual appearance. For example, implementation edges, extension edges, composition edges, etc.

[2]The design document could be viewed on `https://github.com/yingjie-xu/JetUML/blob/Issue375e_EdgeLayoutExperimental/docs/layout/layout.md`

Priority rule Base on the edge category, we establish a priority rule so that edges in one category with higher priority get drawn first. For example, suppose we have extension edges with higher priority than the implementation edges, the extension edges would always be assigned a more centred position compared with implementation edges.

Edge aggregation and node position The edges with the highest priority among all the edges on the same node side would always take the central position as the connection point. All the edges within the same category would be aggregated.

In contrast, for those edges that could not take the central position, we would only aggregate those edges within the same category if they are on the same side of the node. Otherwise, we would separate them to avoid edge crossings.

In figure 2 [3], the extension edge has the highest priority so it takes the central position. The other two implementation edges are on different sides of the target node, so we do not aggregate those two edges even they are within the same edge category.

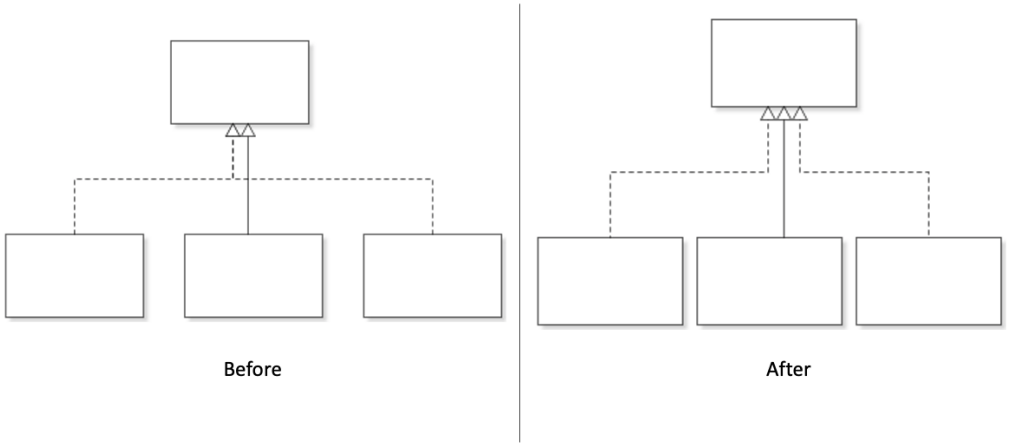

Figure 2: Better way of aggregating edges

[3]The "after" diagram is drawn from this commit: `https://github.com/yingjie-xu/JetUML/commit/fa2511ee983cbcbb83d177d24b8f53652cd1933d`

Optimize the edge path After applying the rules above, we would consider the whole layout of all the edge paths and try to avoid edge crossings by adjusting the turning points on the edge paths. An example of this optimization could be seen in figure 3.

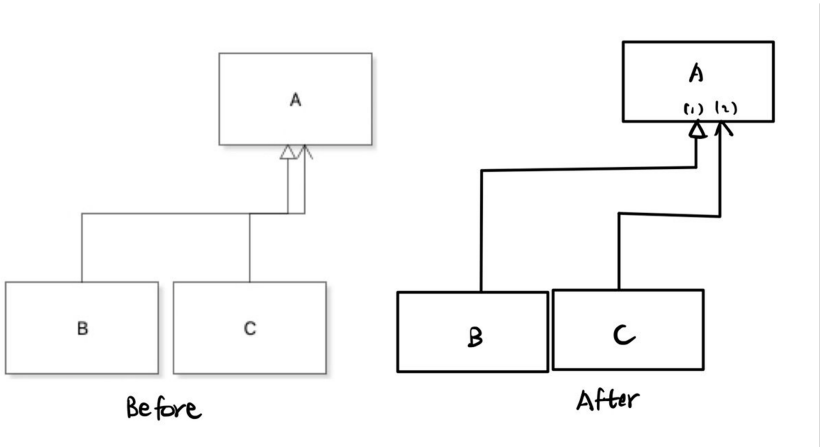

Figure 3: Optimize the edge paths

Stateful drawing design In order to achieve this design, moving away from the stateless edge drawing design would be necessary. In figure 4, we could see the sequence diagram for the stateless design, the getPath method is called for each separate edge and they do not share information with each other. The updated stateful design is covered in section 3.2.

3.2 Layouter design

As described in the previous section, we need a stateful drawing design to achieve the updated algorithm. Layouter is designed to solve this problem. It is an additional layer of indirection between the DiagramViewer and the EdgeViewers. Previously, DiagramViewer would iteratively send each edge to its corresponding EdgeViewer, and EdgeViewers would not only calculate the edge path but also draw the edges. By adding the layouter, we would separate the responsibility of calculating edge paths from EdgeViewers to the layouters. Since the layouter is calculating edge paths for all the edges on the diagram, it is stateful and it allows us to calculate new paths based on previously calculated paths.

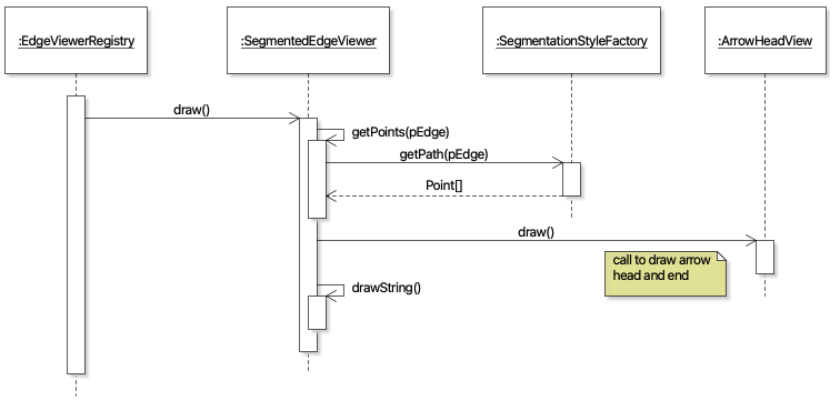

Figure 4: Sequence diagram for previous stateless drawing design

We would use the Use Case diagram as an example in this section. As shown in figure 5, we added an interface named EdgeLayouter. It has a layout method that would take EdgeLayout as a parameter. EdgeLayout is basically a wrapper for the HashMap which uses Edge as a key and its EdgePath as the value. EdgePath is an array of points that represents the edge path. The goal of calling the layout method is to calculate the EdgeLayout for the corresponding diagram. Inside the DiagramViewer, we would create a single instance of the corresponding Layouter.

By calling the draw method inside DiagramViewer (figure 6), we would do two things. The first one is to calculate the edge path by calling the layout method from Layouter. The second one is to find the corresponding EdgeViewer and draw that edge by using the EdgePath calculated from the previous step. Another thing I did not explicitly draw in the sequence diagram is that we could cache the results for getBounds method since we have stored the result inside the EdgeLayout instance.

The stateful layouter design is quite beneficial. It applies the separation of concerns principle. The DiagramViewer could cache the edge layout for future queries. An example of that would be caching the results of the getBounds method in a HashMap. The layouter could take all the edges on the diagram into consideration when planning the new edge paths and avoid as many edge crossings as possible. The EdgeViewer would only need to draw the edges by using the resulting edge paths provided by the layouter.

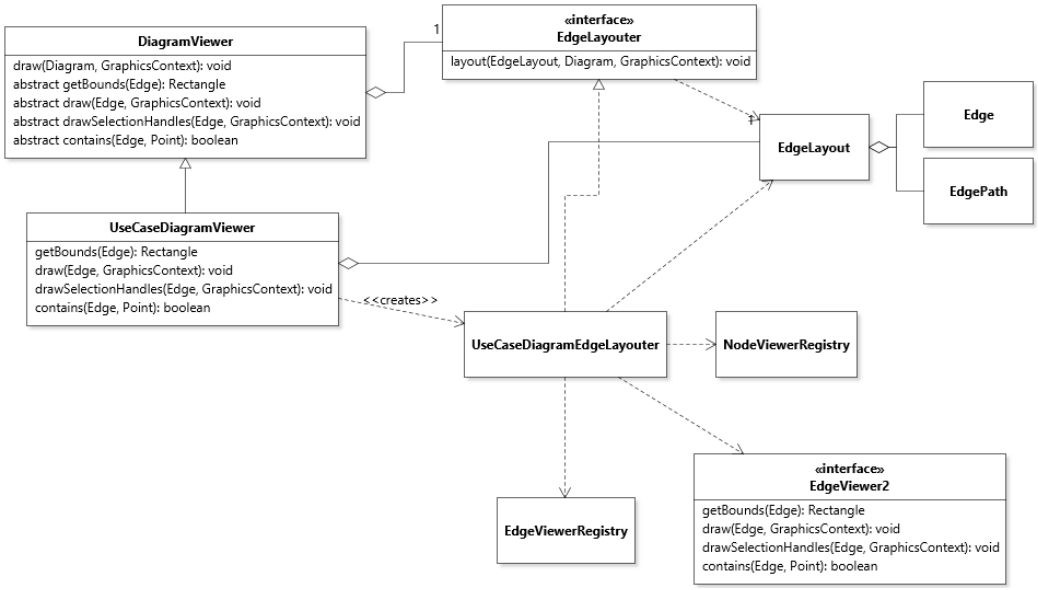

Figure 5: Class diagram for the Use Case diagram Layouter

4 Challenges and Solutions

4.1 State diagram EdgeLayout

The state transition edge is the most important part for transitioning the state diagram to the design with layouter. Originally the state transition edge viewer would not only calculate the edge path but also draw the shape. My initial thought was to move both parts from the edge viewer to the layouter and came up with the design in figure 7.

However, after applying the separation of concerns principle, I came up with the following rules: DiagramViewer should be responsible for calling the corresponding layouter and storing the cached results. Layouter should only be responsible for planning the path and position for all the edges. Although shapes might be considered when calculating the path, the layouter should not return shapes. EdgeViewer should use pre-calculated edge paths to draw the actual edges. As a result, the state diagram layouter would only calculate the path and the state transition edge viewer would use the path to draw the edges.

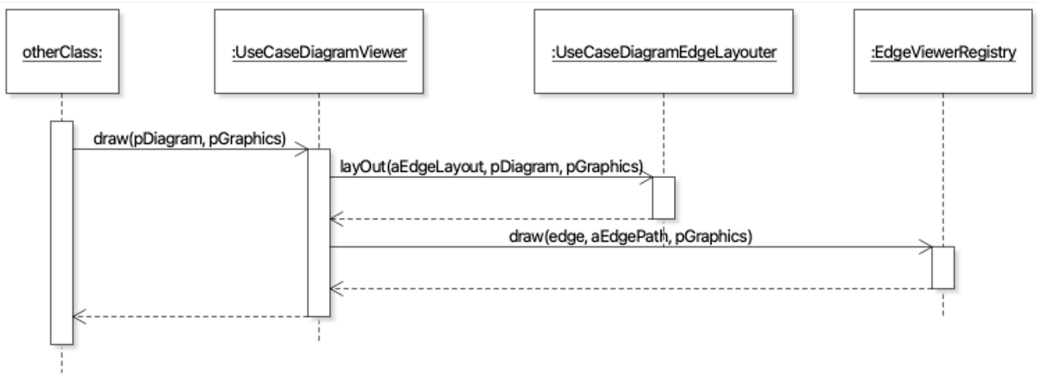

Figure 6: Sequence diagram for the Use Case diagram Layouter

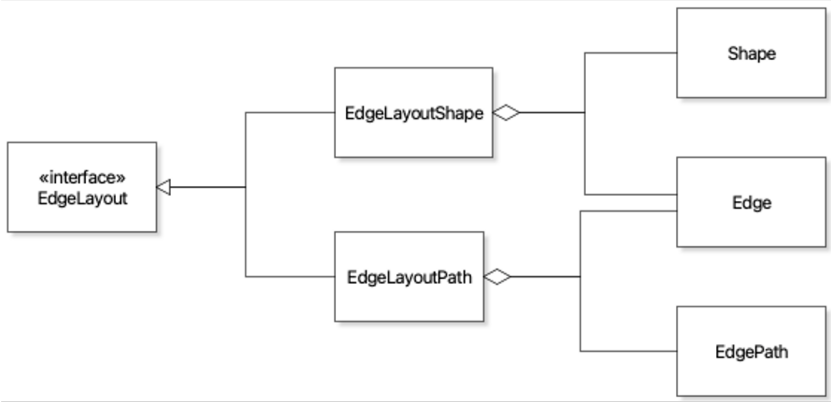

Figure 7: First design for state diagram edge layout

4.2 Position calculation in State diagram

As mentioned in previous sections, we updated the design of drawing edges from stateless to stateful. Position calculation in state diagram would be an example that obtains benefits from this design. The concept of position is used when two StateTransitionEdges have the same start node and the end node. In order to give them different edge paths so that they are not completely overlapped with each other, we give them different position values to separate them. An example of that could be seen in figure 8. In the stateless design, whenever we want to draw a StateTransitionEdge, we need to recalculate its position value and iterate through all the edges in the current diagram. This would result in a lot of duplicated calculations because

we are not able to cache the information anywhere. With our updated design of layouter, I used a nested HashMap to store the number of edges with the same start node and end node. By doing so, we could simply go to the nested HashMap and get the position value without iterating through all the edges. Suppose N is the number of StateTransitionEdges in the diagram, the time complexity for calculating the position value for one StateTransitionEdge would decrease from $O(N)$ to $O(1)$.

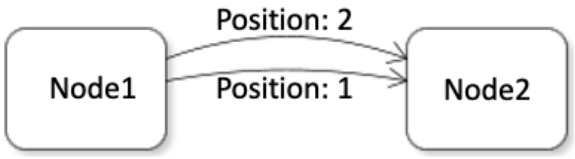

Figure 8: The concept of position in state diagram

4.3 State diagram EdgePath

In the updated design, we calculate the value of position inside Layouter, however, we need to pass this information further to the EdgeViewer so that it could assign different arc degrees to edges with different positions. We have two options to do this, the first one is to store the information somewhere inside EdgePath so that the EdgeViewer could get the position when drawing the edges. The second one is to let the EdgeViewer calculate the information by passing the whole layout into the EdgeViewer.

The first approach I tried is to add position as an attribute in the EdgePath class. However, EdgePath is used for all the diagrams in JetUML and position is only useful for the state diagram. We decided to try other approaches.

The second approach is to pass the whole layout into EdgeViewers and re-calculate the edge position every time. This design would result in passing too much additional information into the EdgeViewers. Similar to the first approach, the whole layout is only used by StateTransitionEdgeViewer and not used by other EdgeViewers.

In the final approach, we decided to add a subclass of EdgePath class named StateDiagramEdgePath and include position as an attribute into this class (figure:9). This design resolves the problems in the previous two approaches and the StateTransitionEdgeViewer could use the StateDiagramEdgePath to obtain the position value.

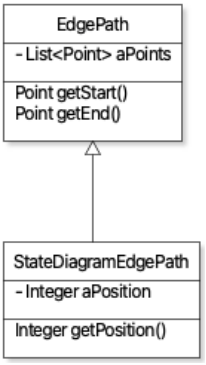

Figure 9: The updated design for storing the position