

# Rapport AP4B: Éditeur de graph interactif

Yann Derré | Esteban Becker | Elise Albrecht | Pierre-Olivier Cayetanot

Juin 2023

## Résumé

Nous avons dû réaliser un projet Java qui contient les fonctionnalités suivantes :

- Edition et sauvegarde d'un graphe à partir d'une description sur fichier.
- Visualisation du graphe dans une fenêtre, choix des couleurs et graphisme, possibilité de zoomer.
- Visualisation interactive des attributs associés aux sommets et aux arrêtes (position, distance, nom de rue par exemple).
- Edition des attributs et paramètres de façon interactive.
- Ajout/suppression interactif de sommets et d'arcs avec clavier/souris en définissant les commandes clavier/souris appropriées.
- Calcul de plus court chemin.

Pour nous organiser et partager le travail, nous avons décidé d'utiliser de placer le code sur GitHub.

Le répo contenant le code source et les dépendances requises pour compiler le projet depuis source est disponible ici [github.com/estebanbecker/AP4B\\_project](https://github.com/estebanbecker/AP4B_project)

# Table des matières

<b>1</b>	<b>Structure du projet</b>	<b>2</b>
<b>2</b>	<b>Cas d'utilisation</b>	<b>4</b>
<b>3</b>	<b>Model</b>	<b>5</b>
3.1	Graph . . . . .	5
3.1.1	Ajout d'information dans le graph . . . . .	6
3.1.2	Récupération d'informations . . . . .	7
3.1.3	Modification du Graph . . . . .	7
3.2	Recherche d'un itinéraire le plus court entre deux points . . . . .	7
<b>4</b>	<b>Contrôleur</b>	<b>9</b>
4.1	Traitement interactions utilisateur . . . . .	10
4.1.1	Gestion du curseur . . . . .	11
4.2	Initialisation de la Vue . . . . .	11
4.3	Interactions avec Modèle . . . . .	12
4.3.1	Fonction <i>findClickedNode()</i> . . . . .	12
4.4	Déplacement dans le graph . . . . .	12
4.4.1	Gestion conflit avec le dragging node . . . . .	13
4.4.2	Hovering node & feedback utilisateur . . . . .	13
4.4.3	<i>Snap to Grid</i> . . . . .	14
4.4.4	Menu contextuel . . . . .	15
4.4.5	Mise à jour . . . . .	15
4.5	Étude de cas : ajout d'un nœud . . . . .	16
4.6	Édition et sauvegarde depuis des fichiers . . . . .	18
<b>5</b>	<b>Vue</b>	<b>19</b>
5.1	Représentation des éléments Graph . . . . .	19

# 1 Structure du projet

Une des contraintes de structure avec lesquelles nous avons dû opérer est l'application d'un modèle MVC pour segmenter les classes du projet.

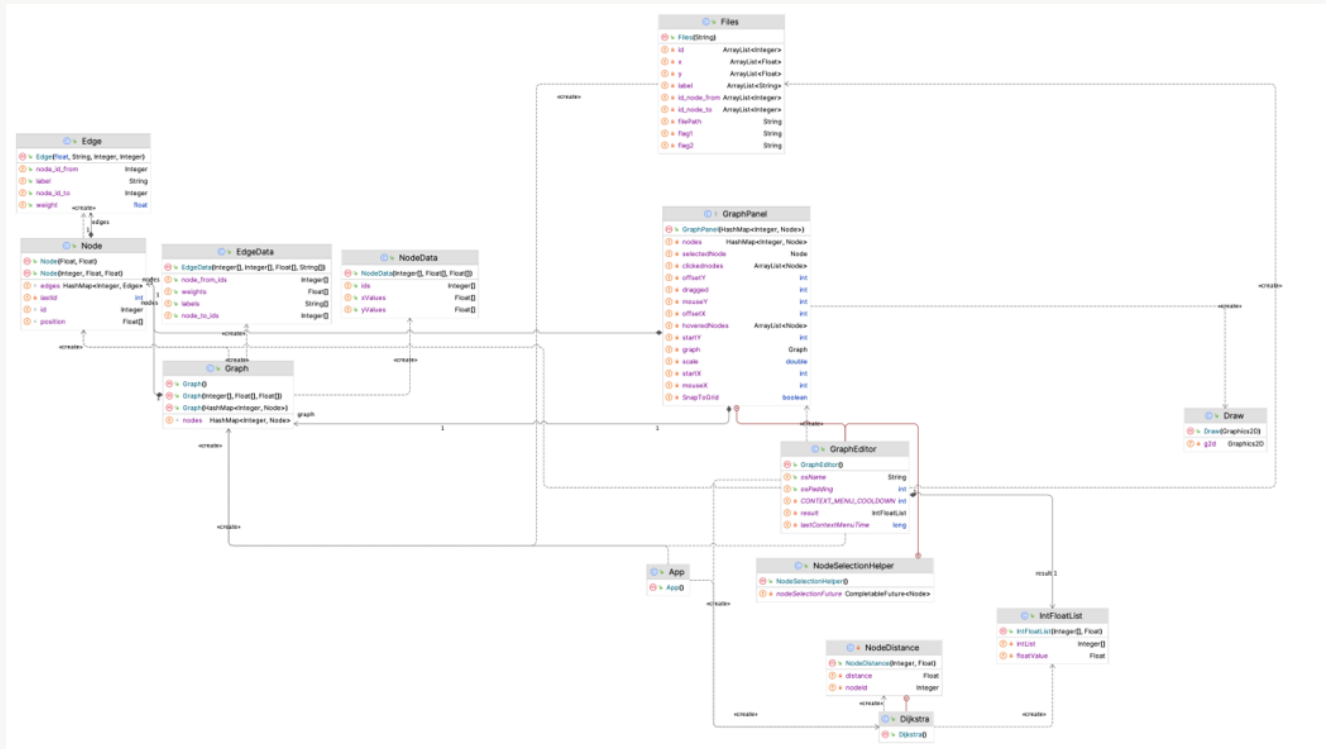


FIGURE 1 – Diagramme de classe du projet

Ci-dessus se trouve le diagramme de classe du projet. On peut identifier 3 sections distinctes, une à gauche, contenant les structures de données du Graphe, une à droite chargée de dessiner les éléments, et une troisième, connecté en tant qu'intermédiaire entre les deux, principalement *GraphPanel()* et *GraphEditor()*

Pour une vision plus claire, voici la version abstraite du modèle MVC implémenté dans notre projet :

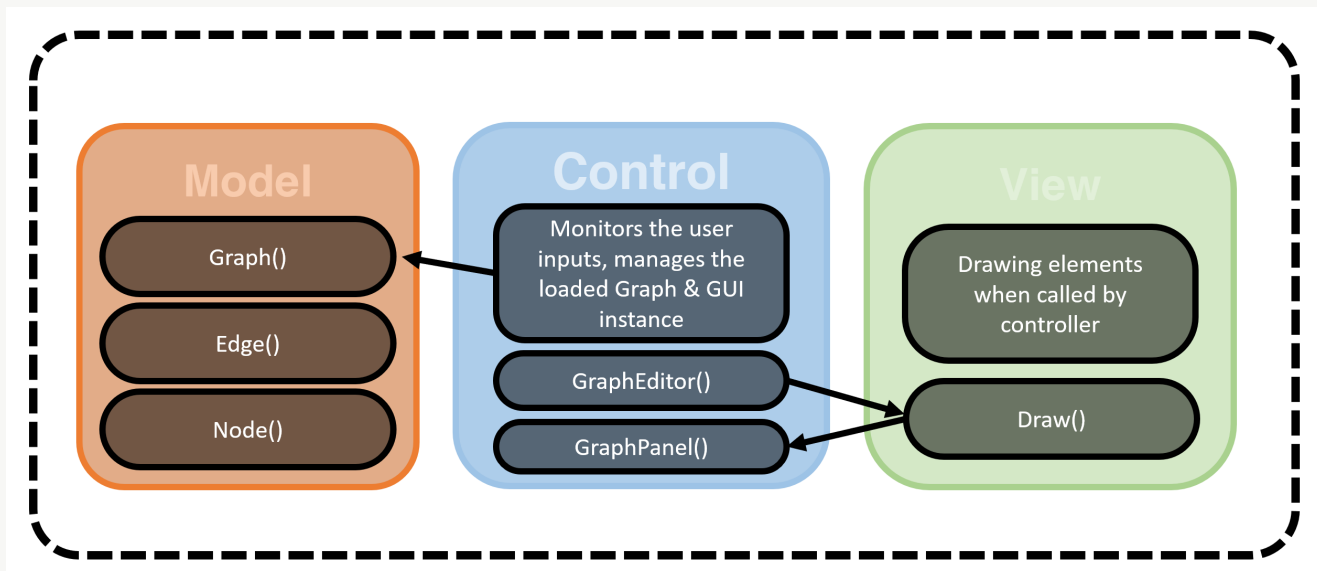


FIGURE 2 – Représentation abstraite du modèle MVC

## 2 Cas d'utilisation

Ainsi, pour définir les actions possibles par l'utilisateur dans l'interface pour manipuler la class Graph, nous avons fait un diagramme cas d'utilisation.

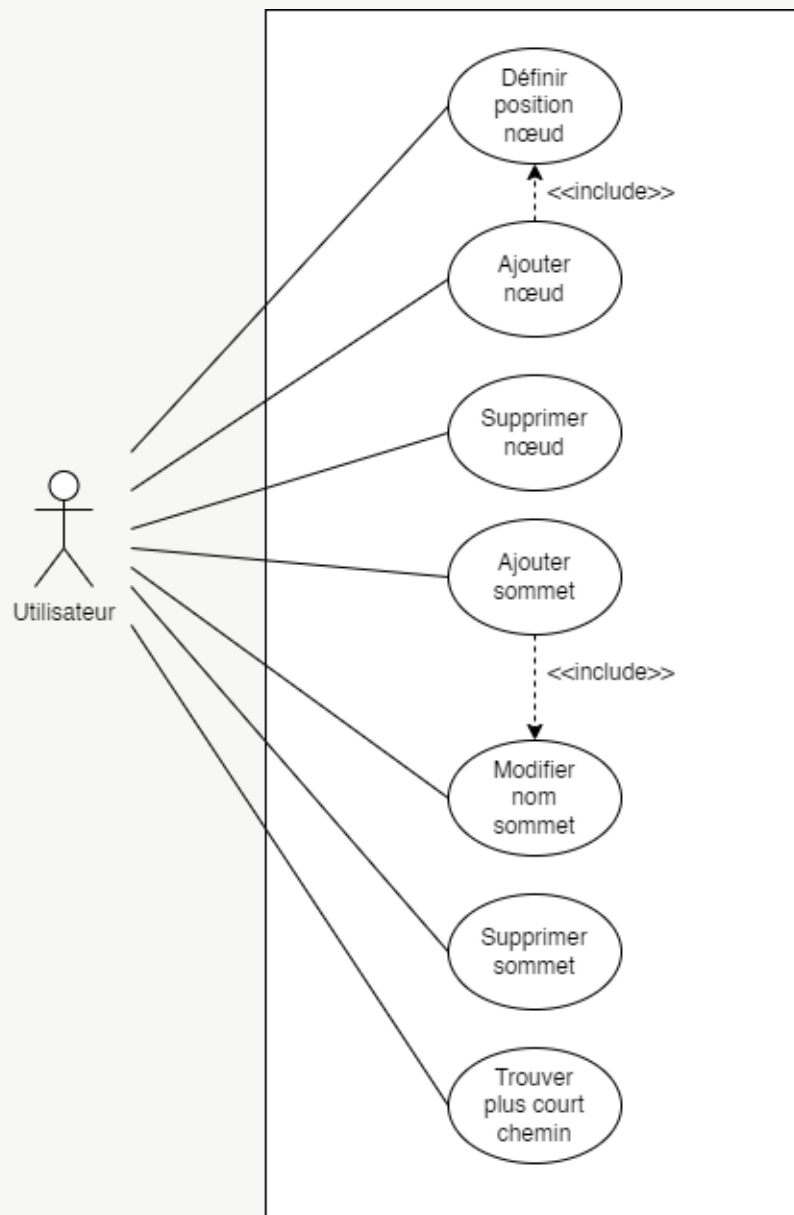


FIGURE 3 – Diagramme de cas d'utilisation de l'interface

## 3 Model

### 3.1 Graph

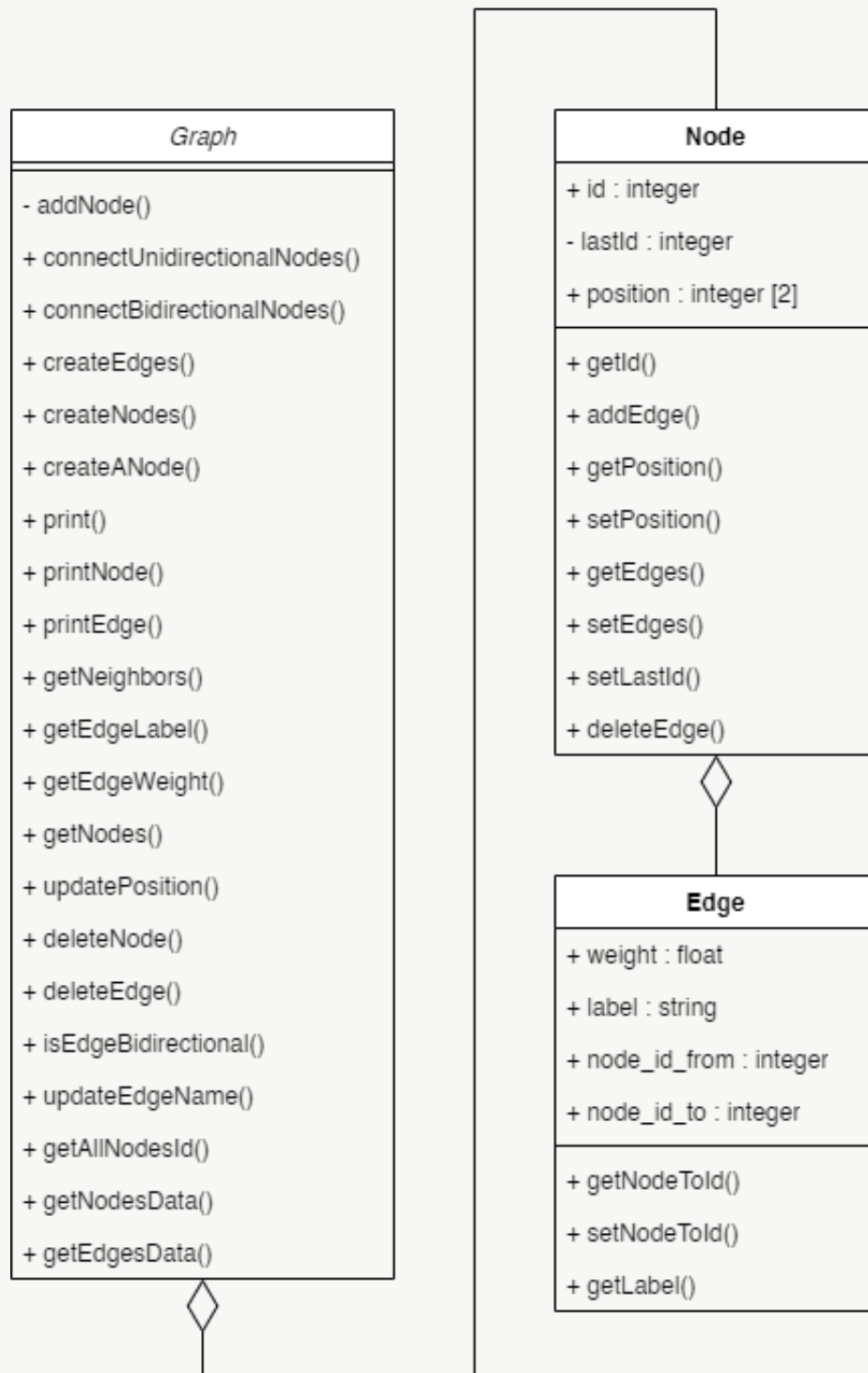


FIGURE 4 – Diagramme de class de la class Graph

Ainsi, la class graph est composé de la class Node qui est composée de la class Edge. Le tout a été mis dans un package Graph. Ceci permet d'importer le package entier dans n'importe quel autre projet vu qu'il est entièrement autonome.

En code, nous l'avons implémenté avec des HashMap :

```
1 HashMap<Integer, Node> nodes;  
  
1 HashMap<Integer, Edge> edges;
```

De plus, tel que déclaré, les HashMap ne sont accessibles que depuis le package. En effet, l'idée est que la complexité des edges et des nodes soient cachés lors de l'importation du package et que l'on utilise uniquement des fonctions de graph.

Nous avons choisi les HashMap car elles permettent de directement relier un id de nœud à un lui-même ou bien un id de Nœud à la liaison qu'il a avec un autre sans les numéroter de façon continue. En termes de code, nous n'avons pas à parcourir toute une liste pour trouver l'objet qui nous intéresse, il suffit d'utiliser la ligne suivant :

```
1 Node node = nodes.get(node_to_id);
```

### 3.1.1 Ajout d'information dans le graph

Dans le cas d'une connexion bidirectionnel entre deux points, nous avons décidé d'utiliser deux Edges, chacun dans un sens.

Afin de simplifier la création d'un graphe (en particulier lors de la lecture, voir : 4.6) un graphe peut être créé avec le constructeur suivante pour créer directement plusieurs nœuds :

```
1 /**  
2  * Initializes a graph with a list of nodes  
3  * @param id      A list of ids  
4  * @param x       A list of x coordinates  
5  * @param y       A list of y coordinates  
6  */  
7  public Graph(Integer[] id, Float[] x, Float[] y)
```

Une fois cette étape créée, si l'on veut créer plusieurs edges simultanément, on peut utiliser la fonction suivante :

```
1 /**  
2  * Creates edges from a list of node ids and labels  
3  * @param node_from_id  The ids of the nodes from which the edges  
start  
4  * @param node_to_id    The ids of the nodes to which the edges end  
5  * @param label          The labels of the edges  
6  */  
7  public void createEdges(Integer[] node_from_id, Integer[] node_to_id,  
String[] label)
```

Afin de connecter deux nœuds, on a accès aux deux fonctions suivantes en fonction de si l'on veut créer une liaison unidirectionnelle ou bidirectionnelle :

```
1 /**  
2  * Connects two nodes with an edge in both directions  
3  * @param node_from_id  The id of the node from which the edge starts  
4  * @param node_to_id    The id of the node to which the edge ends  
5  * @param label          The label of the edge  
6  */  
7  public void connectUnidirectionalNodes(Integer node_from_id, Integer  
node_to_id, String label)  
  
1 /**  
2  * Connects two nodes with an edge in both directions  
3  * @param node1         The id of one node that will be connected  
4  * @param node2         The id of the other node that will be connected  
5  * @param label          The label of the edge  
6  */
```



```

7     public void connectBidirectionalNodes(Integer node1, Integer node2, String
      label)

```

### 3.1.2 Récupération d'informations

Pour obtenir toutes les informations du graphe, la fonction suivante parcourt itérativement le graph pour afficher tous les nœuds et toutes les liaisons :

```

1     /**
2      * Prints the graph
3      */
4     public void print() {
5         for (Node node : nodes.values()) {
6             System.out.println("Node " + node.getId() + " at position (" +
node.position[0] + ", " + node.position[1]
7                 + ") has edges:");
8             for (Edge edge : node.edges.values()) {
9                 System.out.println("    Connected with Node: " + edge.
node_id_to + " with label " + edge.label
10                    + " and weight " + edge.weight);
11             }
12         }
13     }

```

De plus, on peut demander à la structure graphe :

- Si une liaison existe dans les deux directions.
- La liste de tous les id des nœuds. En effet, lors de la création, ils se suivent, mais une fois un nœud supprimé, ils peuvent ne plus se suivre.
- Exporter les informations de tous les nœuds dans une structure NodeData créé pour l'occasion. (Particulièrement utile pour l'enregistrement)
- De même pour les edges avec une structure EdgeData
- Liste des voisins d'un nœud. Attention, cette fonction ne donne que les nœuds en suivant les flèches du graph.

### 3.1.3 Modification du Graph

Il existe une fonction pour mettre à jour des éléments du graph pour chacune des informations qu'il possède :

- Mettre à jour la position d'un nœud. Le Graph étant euclidien, il faut penser à mettre à jour la position de chaque Edge où le graph est connecté. Cette étape manque d'optimisation, en effet, on est obligé de parcourir tout le graphe pour savoir à quoi il est connecté. En effet, s'il y a la liaison 2 -> 3. En lisant les informations du nœud 3, on ne sait pas qu'il est connecté à 2.
- Mettre à jour le nom d'un edge.
- Suppression d'un élément du graphe :
  - Un Edge
  - Un nœud, cette fonction possède un second paramètre «force». Si un nœud possède encore des edges, il ne sera pas supprimé sauf si force=true.

## 3.2 Recherche d'un itinéraire le plus court entre deux points

Pour trouver l'itinéraire le plus court, nous avons utilisé au choix l'algorithme de Dijkstra.

Il s'agit d'un algorithme de programmation dynamique. En effet, pour trouver le plus court chemin entre le départ et l'arrivée, on cherche le plus court chemin entre le départ et les nœuds

précédents de l'arrivée, puis les nœuds précédents. Ainsi, l'algorithme de Dijkstra utilise une liste de priorité pour à chaque fois explorer le nœud avec la valeur la plus faible. Cette structure de donnée est codée ainsi :

```
1    PriorityQueue<NodeDistance> queue = new PriorityQueue<>();
```

Ainsi, pour ajouter un élément ou récupérer l'élément prioritaire, on utilise :

```
1    queue.add(node)
2    node = queue.poll()
```

Nous arrêtons la recherche quand le nœud en cours d'exploration est présent dans les nœuds déjà explorés en sens inverse. Pour stocker la liste des nœuds à visiter par ordre de priorité (temps le plus court pour y accéder) nous utilisons la structure du tas qui permet d'accéder rapidement à l'élément ayant la plus petite valeur.

## 4 Contrôleur

Le contrôleur est le module du projet chargé d'agir en tant qu'intermédiaire entre l'utilisateur, interagissant avec le modèle pour relayer ses actions, puis les afficher sur la Vue.

De par sa nature, le contrôleur et la vue sont liés de manière proche : voici ci-dessous le diagramme de classe du Contrôleur et Vue.

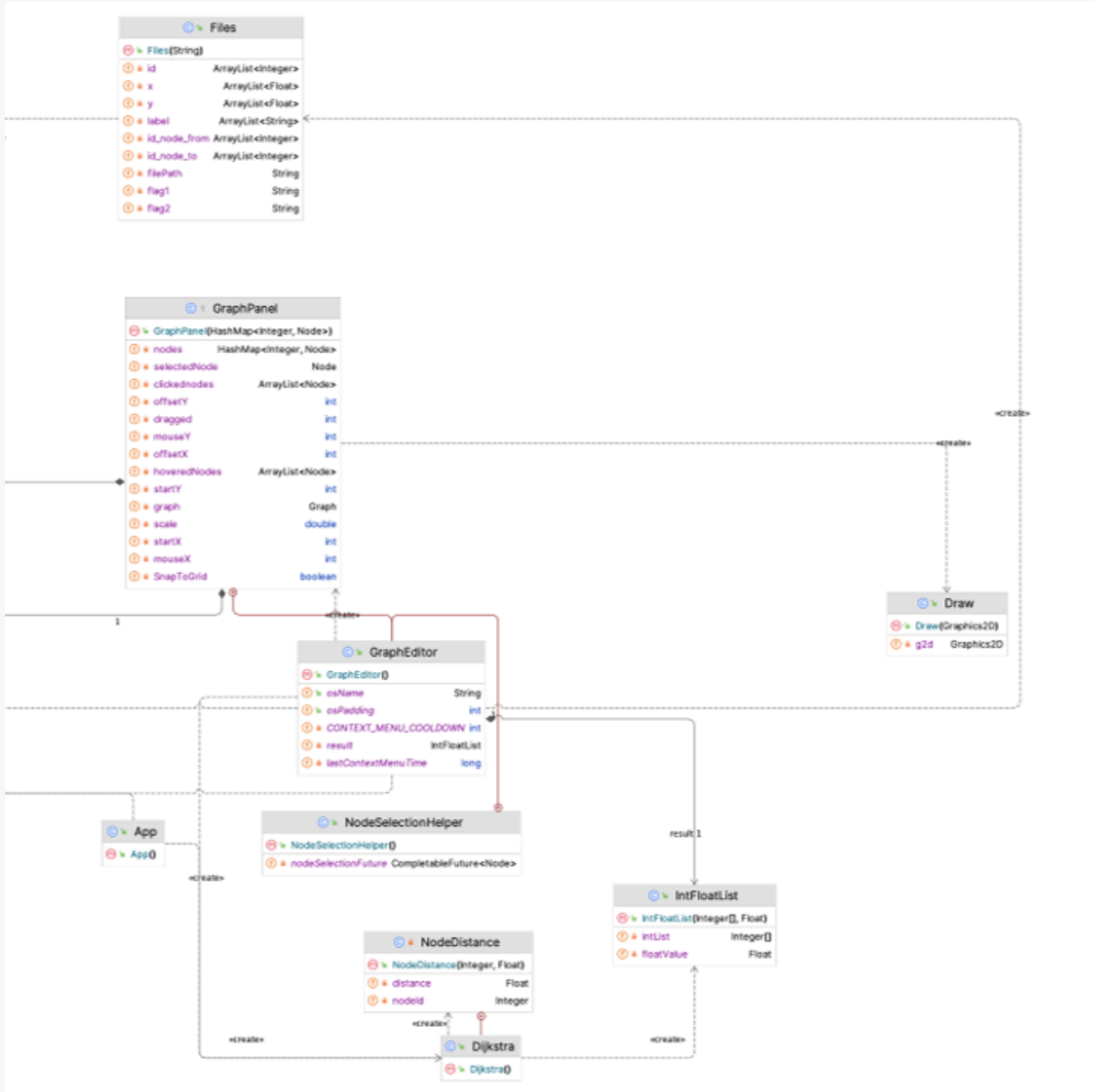


FIGURE 5 – Diagramme de classe du Contrôleur + Vue

Le contrôleur est contenu dans la classe GraphEditor. Elle se compose de 2 éléments majeurs :

```

1 public static void createAndShowGUI(Graph graph);
2 protected static class GraphPanel extends JPanel;

```

Le premier possède une fonction explicite, il va s'occuper d'initialiser le conteneur graphique **GraphPanel()** sur lequel la Vue va pouvoir dessiner et représenter le graph. Les éléments statiques de UI, tels que le FAB (floating action button) permettant l'action principale d'ajouter un nœud, ou la *menu bar* permanente permettant d'ouvrir/sauvegarder les graphes est également

définie ici.

Voici une vue des éléments générés par le contrôleur

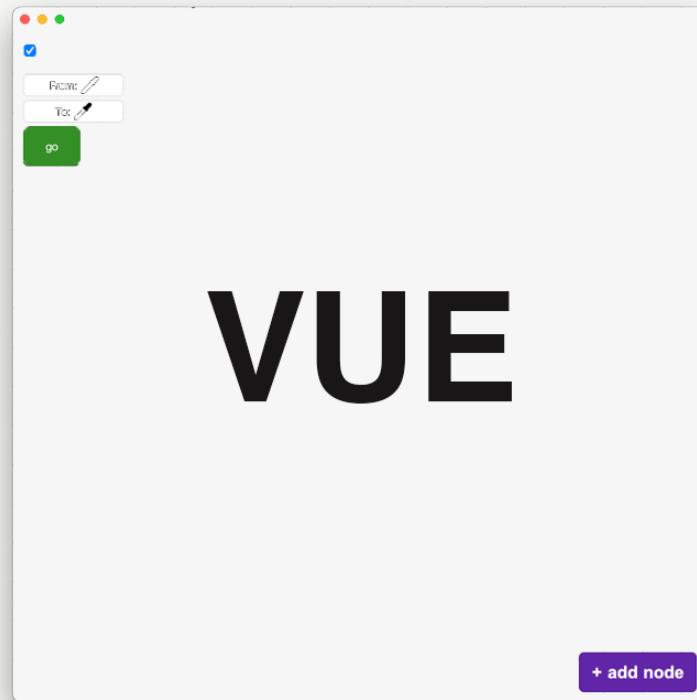


FIGURE 6 – Aperçu de la UI

**GraphPanel()**, lui, va s'occuper de gérer les *inputs* utilisateurs : en ajoutant des *EventsListeners*, il va pouvoir intercepter le curseur utilisateur pour modifier le comportement de la Vue.

#### 4.1 Traitement interactions utilisateur

Les interactions utilisateur sont gérées par **GraphEditor()**

Dans l'objectif de créer une application avec des contrôles à ergonomie moderne, nous devons gérer les mouvements auxquels l'utilisateur s'attend à pouvoir effectuer :

- **Pan** : se déplacer dans le graphe en maintenant son bouton principal
- **Pinch/Zoom** : pouvoir zoomer confortablement sur une section du graphe en *scrollant* sur sa molette/trackpad
- **Sélectionner les noeuds** : en cliquant sur un noeud, l'utilisateur s'attend à pouvoir le sélectionner, un indicateur doit refléter cette action sur le noeud
- **Connecter des noeuds** : après avoir sélectionné un noeud, l'utilisateur va vouloir cliquer sur un second pour les relier ensemble.
- **Déplacer les noeuds** : en recevant un feedback que le noeud est interactif, l'utilisateur va vouloir déplacer le noeud en maintenant son click principal enfoncé. Ce mouvement ne doit pas rentrer en conflit avec l'action de *panning* de la Vue.
- **Édition du noeud/edge** : pour accéder aux informations et options de modification avancées du noeud, l'utilisateur s'attend à pouvoir utiliser son click secondaire sur un noeud.

- **Création précise d'un noeud** : l'utilisateur s'attend à pouvoir créer un noeud aligné à la grille/placer son noeud sur un point intermédiaire, l'interface doit lui permettre de placer son noeud au point attendu

Nous avons choisi d'opter sur une démarche minimaliste pour le design de la UI. En se reposant sur les comportements déjà inculqués à l'utilisateur, nous n'avons pas besoin de dessiner toutes les actions que l'utilisateur peut faire en même temps.

#### 4.1.1 Gestion du curseur

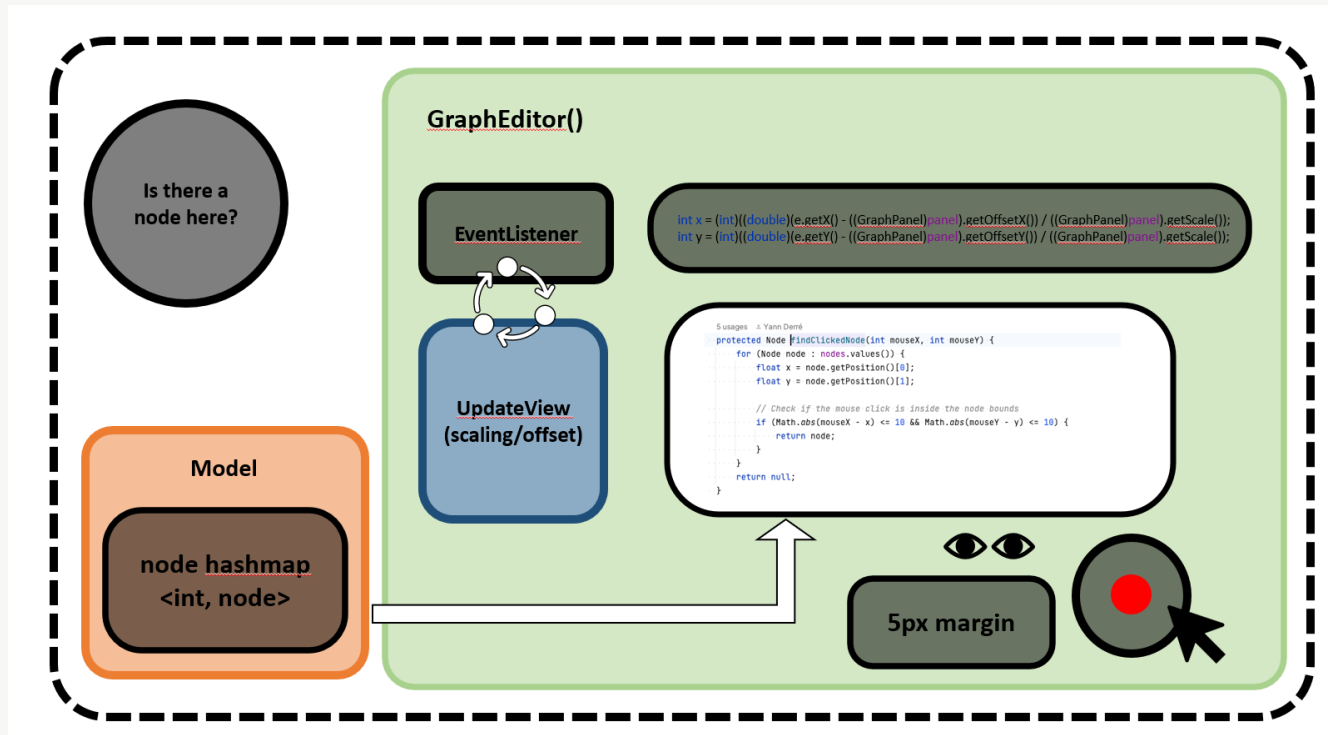


FIGURE 7 – Conversion des coordonnées du pointeur pour interaction dans le Graphe

Une approche naïve serait de prendre les coordonnées (x,y) obtenues directement depuis un *MouseListener* et les utiliser directement. Cependant, notre application permet de se déplacer et de zoomer dans la vue.

En réalité, un *MouseListener* ne peut voir que les coordonnées du pointeur en fonction de la fenêtre globale (voir Fig.6), pas du contenu intérieur. Il est alors nécessaire de convertir les coordonnées obtenues pour aller de ce repère à un repère relatif/local, correspondant à la vue de l'utilisateur.

## 4.2 Initialisation de la Vue

Dans sa fonction **paintComponent()**, le contrôleur initialise le conteneur graphique, et le passe à la vue

```

1      @Override
2      protected void paintComponent(Graphics g) {
3          Graphics2D g2d = (Graphics2D) g;
4          Draw draw = new Draw(g2d);

```

Puisque le contrôleur se charge d'initialiser le conteneur graphique sur lequel la Vue dessine, il se charge également d'appliquer les déformations graphiques au résultat du dessin de la vue pour permettre le zoom et le déplacement

```
1      // Apply zoom and offset transformations
2      g2d.translate(offsetX, offsetY);
3      g2d.scale(scale, scale);
```

## 4.3 Interactions avec Modèle

### 4.3.1 Fonction *findClickedNode()*

Un des composants majeurs pour l'intention utilisateur au modèle est la fonction **findClickedNode()** : elle permet de retrouver l'objet noeud dans la hashmap du modèle correspondant à l'élément graphique cliqué par l'utilisateur.

Comme vu précédemment, nous pouvons convertir les coordonnées du pointeur en fonction de la fenêtre du programme en coordonnées relatives à la vue du graphe affichée actuellement sur l'écran en tenant compte de l'offset et du zoom appliqué.

```
1
2 protected Node findClickedNode(int mouseX, int mouseY) {
3     for (Node node : nodes.values()) {
4         float x = node.getPosition()[0];
5         float y = node.getPosition()[1];
6
7         // Check if the mouse click is inside the node bounds
8         if (Math.abs(mouseX - x) <= 10 && Math.abs(mouseY - y) <= 10)
9             return node;
10    }
11    }
12    return null;
13 }
```

La fonction va interagir avec la HashMap du modèle pour voir si la position relative du curseur se trouve dans un rayon de 5 px du noeud. Si un noeud est trouvé dans ce rayon, l'objet nous est renvoyé.

En général, le contrôleur va utiliser les fonctions suivantes pour interagir avec le modèle :

- **graph.deleteNode()**
- **graph.addNode()**
- **graph.updatePosition()**
- **graph.connectUnidirectionalNodes()**
- **graph.updateEdgeWeight()**
- **graph.updateEdgeName()**
- **graph.deleteEdge()**

Nous nous concentrerons sur la **création d'un noeud** dans une étude de cas ci-dessous.

## 4.4 Déplacement dans le graph

Afin de se déplacer dans le Graph, on applique une déformation *offset/scale* sur l'espace 2D sur lequel la Vue dessine. Pour cela, on récupère la position initiale du curseur lorsque le click principal est maintenu, et on calcule le delta entre la position où le click s'est produit initialement et la position actuelle avec click maintenu

```

1      addMouseMotionListener(new MouseMotionAdapter() {
2
3          public void mouseDragged(MouseEvent e) {
4              if (dragged != 1) {
5
6                  int deltaX = e.getX() - startX;
7                  int deltaY = e.getY() - startY;
8
9                  offsetX += deltaX;
10                 offsetY += deltaY;
11
12                 startX = e.getX();
13                 startY = e.getY();
14
15                 repaint();
16             }
17         }
18     });
19
20
21     @Override
22     protected void paintComponent(Graphics g) {
23         //...
24         // Apply zoom and offset transformations
25         g2d.translate(offsetX, offsetY);
26         g2d.scale(scale, scale);
27         //....

```

Notre fonction de mise à jour va ensuite appliquer la déformation graphique au conteneur 2D.

#### 4.4.1 Gestion conflit avec le dragging node

Cependant, cette fonction rentre en conflit avec le comportement attendu par l'utilisateur de pouvoir déplacer les noeuds.

```

1      public void mouseMoved(MouseEvent e) {
2          // Check if a node is clicked and assign it to
3      selectedNode
4
5          selectedNode = findClickedNode(mouseX, mouseY);
6          big = selectedNode;
7          setDragged(0);
8      }

```

Lorsque l'utilisateur survole un noeud, ses coordonnées sont récupérées, si le dragging se produit alors que *big* contient un noeud valide, on désactive le déplacement de la vue en plaçant le booléen sur *True*.

Sinon, on réactive le déplacement

```

1      public void mouseDragged(MouseEvent e) {
2          if (big != null) {
3              // setDragged to 1
4              setDragged(1);
5          }
6          //....
7      }
8

```

#### 4.4.2 Hovering node & feedback utilisateur

Une des missions principales derrière le design de l'interface est de produire un indice clair à l'utilisateur lorsqu'un élément est interactif. De manière analogue à la gestion du conflit dé-

placement vue/node, on identifie lorsque le curseur survole un noeud, et on l'ajoute à l'array `hoveredNodes`.

```

1      public void mouseMoved(MouseEvent e) {
2
3          mouseX = (int) ((e.getX() - offsetX) / scale);
4          mouseY = (int) ((e.getY() - offsetY) / scale);
5
6          Node hoveredNode = findClickedNode(mouseX, mouseY);
7          // if hovered node, make it bigger and show name
8          // if not, remove the bigger node and name
9          if (hoveredNode != null) {
10             hoveredNodes.add(hoveredNode);
11             // System.out.println("hovered" + hoveredNode);
12             // Set the cursor to the hand cursor
13             if (getCursor().getType() != Cursor.CROSSHAIR_CURSOR)
14             {
15                 setCursor(new Cursor(Cursor.HAND_CURSOR));
16             }
17             } else {
18                 // if cursor is cross hair, set it to default
19                 if (getCursor().getType() != Cursor.CROSSHAIR_CURSOR)
20                 {
21                     setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
22                 }
23                 // Set the cursor to the move cursor
24             }
25             repaint();
26         }
27     }

```

La fonction permettant la mise à jour de l'interface va ensuite dessiner l'indicateur sur tous les noeuds survolés.

```

1 protected void paintComponent(Graphics g) {
2     ///...
3     draw.node_circle(hoveredNodes);
4     hoveredNodes.clear();
5     ///...
6 }

```

Ce mécanisme est identique pour les noeuds sélectionnés, nous ne détaillerons que cette version.

#### 4.4.3 *Snap to Grid*

Notre grille représente une incrémentation de 50 pixels. L'utilisateur va être naturellement poussé à placer ses points sur l'alignement. S'il le souhaite, l'option *Snap to Grid* permet d'arrondir la coordonnée du noeud au multiple de 50 le plus proche.

```

1 if (selected){
2     x = (x + 25) / 50 * 50;
3     y = (y + 25) / 50 * 50;
4 }

```

Le contrôleur crée un *ActionListener*, impactant une variable booléenne retransmise aux fonctions d'ajout et de déplacement des noeuds.

```

1 snap.addActionListener(e -> {
2     if (e.getSource() == snap) {
3         panel.setSnap(snap.isSelected());
4         panel.repaint();
5     }
6 });

```



#### 4.4.4 Menu contextuel

Le menu contextuel nous permet de cacher des options d'édition derrière un comportement attendu par l'utilisateur : en cliquant sur un noeud avec son click secondaire, l'utilisateur accède aux attributs et options d'édition du noeud et edge(s) originant de ce dernier (modification label, suppression). S'agissant d'un élément UI statique, ce dernier est dessiné par le contrôleur, et fonctionne de manière analogue aux autres fonctions d'interactions avec les noeuds :

- **MouseListener() dans GraphPanel()** : Lorsque le click secondaire est détecté, les coordonnées sont passées à la fonction **createContextMenu()**
- **Check si noeud présent** : Si **findClickedNode()** nous retourne un noeud valide à la position du curseur, on peut continuer la création du menu.

```
1 public void mousePressed(MouseEvent e) {
2     startX = e.getX();
3     startY = e.getY();
4     mouseX = (int) ((startX - offsetX) / scale);
5     mouseY = (int) ((startY - offsetY) / scale);
6
7     if (SwingUtilities.isRightMouseButton(e)) {
8         createContextMenu(mouseX, mouseY, startX, startY);
9     }
10
11 private void createContextMenu(int mouseX, int mouseY, int X, int Y) {
12     // Check if the context menu was invoked recently
13     if (System.currentTimeMillis() - lastContextMenuTime <
14         CONTEXT_MENU_COOLDOWN) {
15         ContextMenuTime = System.currentTimeMillis();
16         Node clickedNode = findClickedNode(mouseX, mouseY);
17         if (clickedNode == null) {
18             return;
19         }
20     }
21     //...
```

#### 4.4.5 Mise à jour

La fonction **paintComponent()** est en réalité une méthode abstraite obtenue via JPanel. Elle est automatiquement appelée par le contrôleur à chaque rafraîchissement de frame. Elle permet ainsi de contacter la Vue et de lui demander de dessiner une nouvelle frame pour chaque élément lorsque demandé.

```
1
2 @Override
3     protected void paintComponent(Graphics g) {
4         GraphicsEditor();
5         super.paintComponent(g);
6         Graphics2D g2d = (Graphics2D) g;
7         g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
8             RenderingHints.VALUE_ANTIALIAS_ON);
9
10        // Apply zoom and offset transformations
11        g2d.translate(offsetX, offsetY);
12        g2d.scale(scale, scale);
13
14        Draw draw = new Draw(g2d);
15
16        g2d.setStroke(new BasicStroke(2.0f));
```

```

17
18     draw.grid();
19     draw.origin();
20     draw.all_arrows(nodes);
21     draw.nodes(nodes);
22     draw.node_circle(hoveredNodes);
23     draw.node_circle(clickednodes);
24     hoveredNodes.clear();
25
26     if (result != null) {
27         draw.path(result, nodes);
28     }
29
30     g2d.scale(1.0 / scale, 1.0 / scale);
31     g2d.translate(-offsetX, -offsetY);
32 }

```

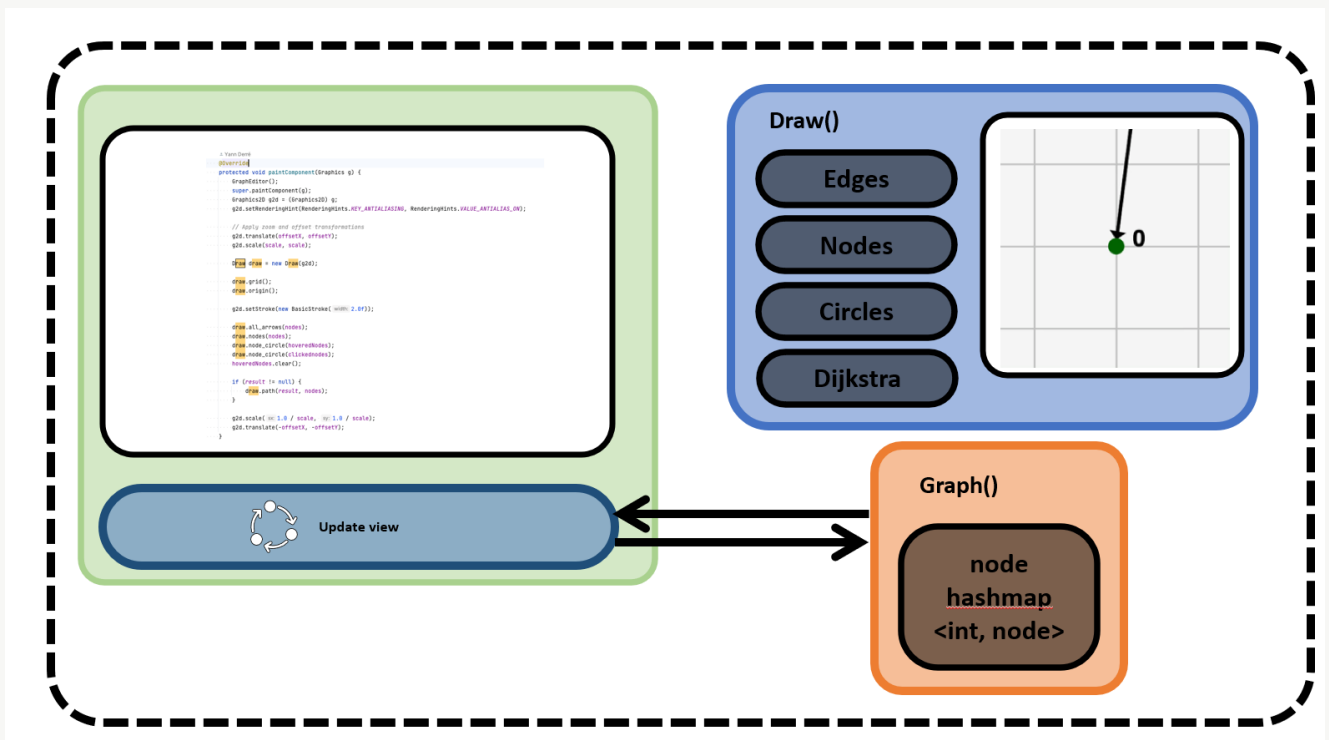


FIGURE 8 – Interaction avec le modèle pour mettre à jour la Vue

## 4.5 Étude de cas : ajout d'un nœud

Maintenant que nous connaissons les mécanismes d'interactions avec l'interface, on peut comprendre comment un utilisateur peut ajouter un nœud avec son curseur.

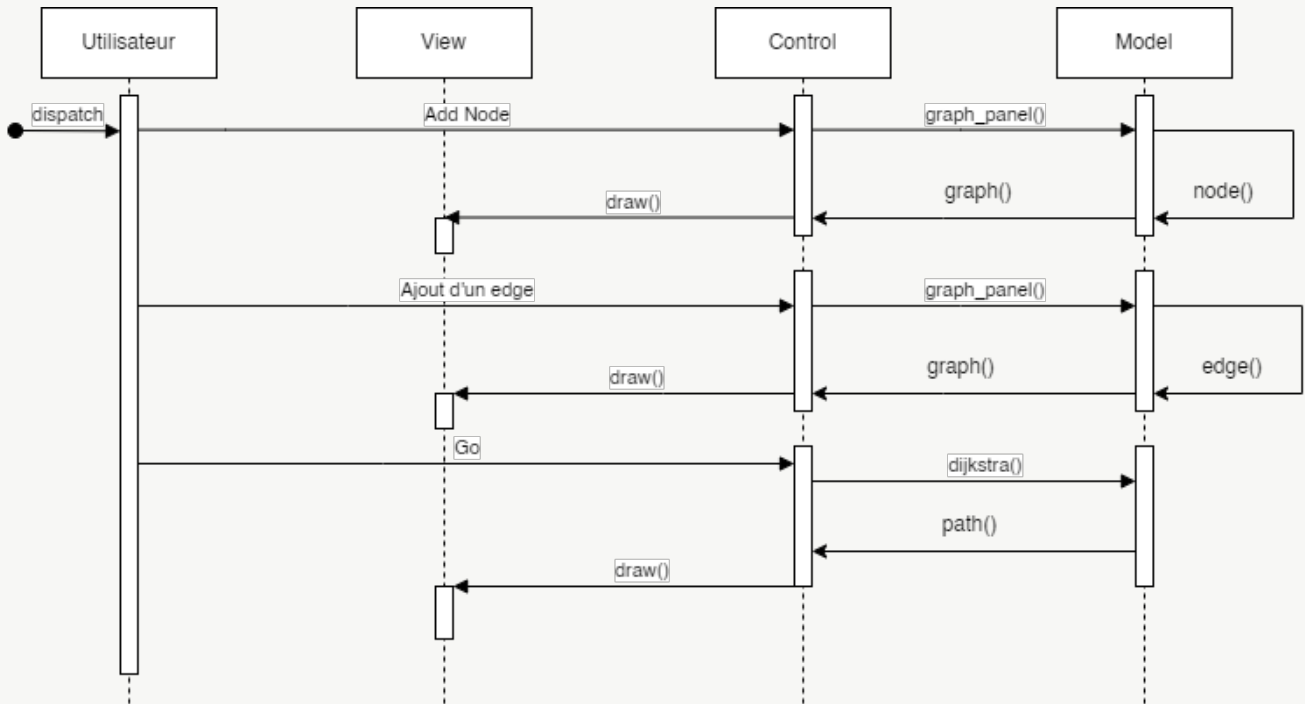


FIGURE 9 – Diagramme de séquence d'ajout d'un noeud/edge

Lorsque l'ActionListener du FAB est activé (bouton pressé) à l'intérieur de **createAndShowGUI()**, il appelle **addNodeonClick()**

```

1 fab.addActionListener(e -> {
2     if (e.getSource() == fab) {
3         addNodeOnClick(panel, graph, snap.isSelected());
4     }
5 });

```

Ce dernier va changer l'apparence du pointeur en *crosshair*, permettant une sélection précise, puis va calculer les coordonnées relatives du curseur à la vue. Lorsqu'un click est détecté sur le *GraphPanel()* via un *MouseListener()*, on peut ajouter dans le modèle un nouveau node aux coordonnées du curseur lors du click.

```

1 // Change cursor to crosshair
2 panel.setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
3 panel.addMouseListener(new MouseAdapter() {
4     @Override
5     public void mouseReleased(MouseEvent e) {
6         if (SwingUtilities.isLeftMouseButton(e)) {
7
8             super.mouseClicked(e);
9             int x = (int) ((e.getX() - ((GraphPanel) panel).getOffsetX
10 ()) / ((GraphPanel) panel).getScale());
11             int y = (int) ((e.getY() - ((GraphPanel) panel).getOffsetY
12 ()) / ((GraphPanel) panel).getScale());
13
14             if (selected) {
15                 x = (x + 25) / 50 * 50;
16                 y = (y + 25) / 50 * 50;
17             }
18             graph.createANode((float) x, (float) y);
19
20             // Repaint the panel
21             panel.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
22             panel.removeMouseListener(this);
23             panel.repaint();
24         }
25     }
26 });

```

```

22         }
23     }
24 });

```

Le *MouseListener* est ensuite détruit et la vue redessinée pour afficher l'impact de l'action.

## 4.6 Édition et sauvegarde depuis des fichiers

Le contrôleur se charge également s'appeler les fonctions de gestion fichier. Puisque ce dernier initialise l'objet Graph représentant le modèle, nous pouvons facilement réinitialiser un contrôleur pour une nouvelle instance du graphe.

Le contrôleur permet de sauvegarder/charger le Graph existant, ou bien d'en créer un nouveau.

```

1  public static void restartProgram(Graph newGraph) {
2      // Update the current graph with the newGraph
3      SwingUtilities.invokeLater(() -> {
4          createAndShowGUI(newGraph);
5      });
6  }

```

L'avantage d'avoir utilise le framework Swing pour concevoir la UI est que l'interface est dessinée sur son propre thread, séparée des autres opérations du projet via *invokeLater()*. Nous utilisons cette fonction pour gérer une nouvelle instance

```

1  frame.dispose();

```

Nous supprimons l'ancienne instance lors du chargement.

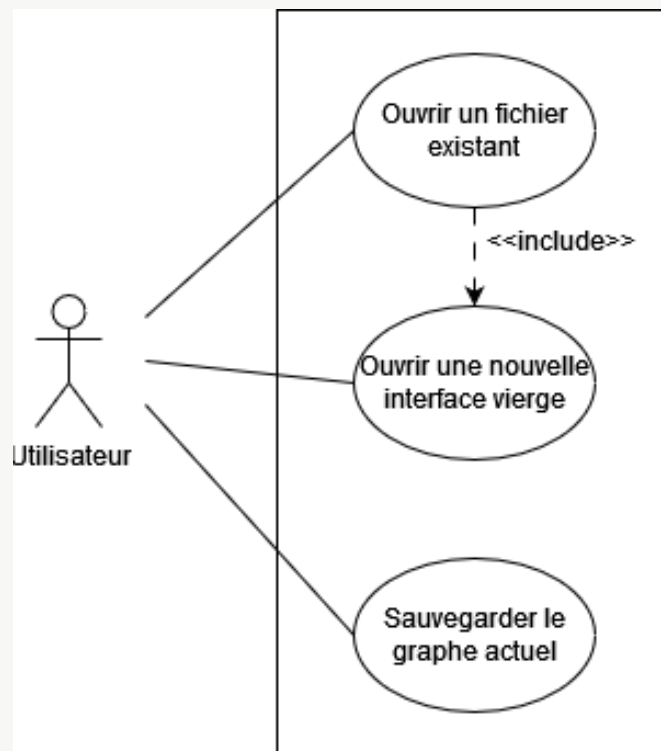
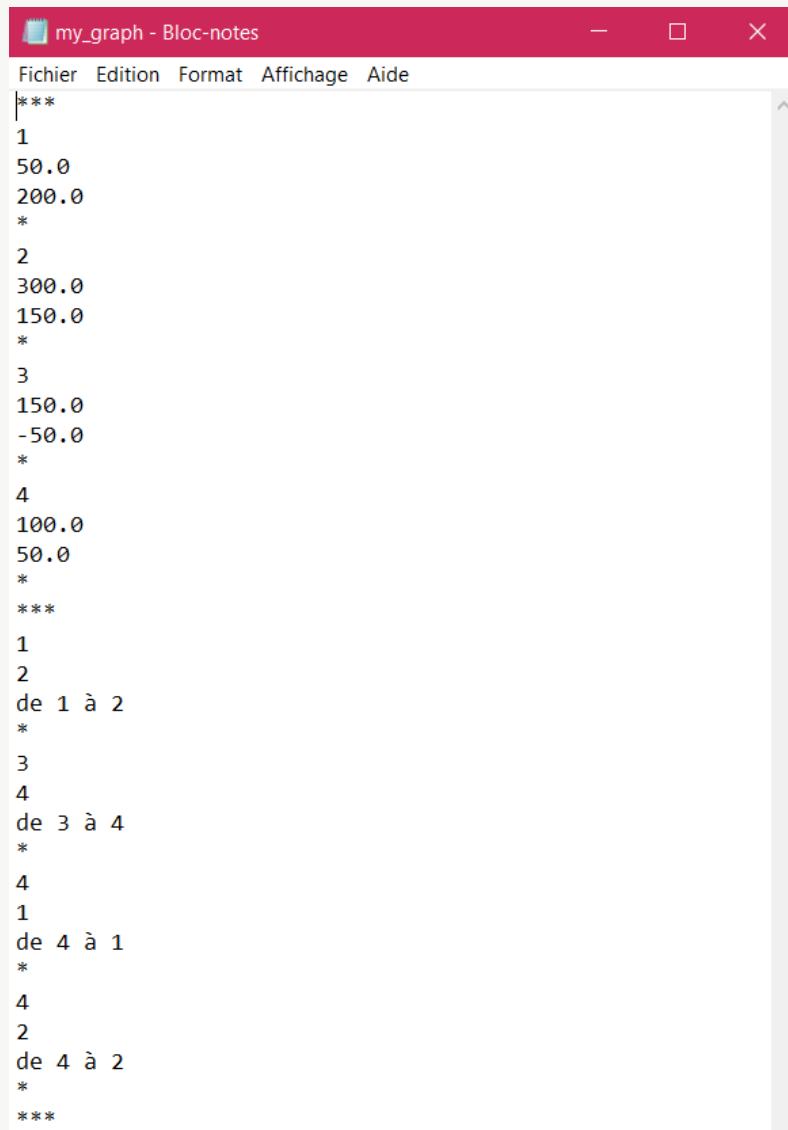


FIGURE 10 – Diagramme des cas d'utilisation des fichiers

Le format de fichier choisi est le .txt pour pouvoir éditer un fichier et le charger ensuite, comme demandé en cours. Le fichier a un formatage particulier et les informations sont scindées en 2 : on a d'abord les informations des nœuds, avec le numéro d'identifiant, puis les coordonnées x et y. Ensuite, on a les informations des arêtes, avec l'identifiant du nœud de départ, celui du nœud d'arrivée et le label associé. Ces informations sont délimitées par des drapeaux : "\*\*\*" pour le

début, la limitation entre les 2 types de données et la fin et "\*" entre chaque groupe de données (nœud ou arêtes).



```
***
1
50.0
200.0
*
2
300.0
150.0
*
3
150.0
-50.0
*
4
100.0
50.0
*
***
1
2
de 1 à 2
*
3
4
de 3 à 4
*
4
1
de 4 à 1
*
4
2
de 4 à 2
*
***
```

FIGURE 11 – Exemple d'un fichier

Pour éditer un fichier, on doit commencer par lire ce dernier, ce qui est réalisé avec la fonction "readFile" de la classe Files. Cette fonction lit un fichier texte contenant des informations sur un graphe, extrait ces informations et crée un objet Graph représentant le graphe avec les nœuds et les arêtes correspondants.

Pour lire un fichier, on fait appel à la fonction "writeFile" qui prend un Graph en paramètres, et crée un fichier avec les informations de ce dernier, en respectant le formatage décrit précédemment.

## 5 Vue

### 5.1 Représentation des éléments Graph

- **Noeud** : représente un sommet du graphe, avec son label associé
  - Rond
  - Label
  - Rond de sélection : il se dessine lorsque l'utilisateur survole le noeud/le sélectionne.
- **Edge** : la connexion entre deux nœuds, elle doit indiquer la direction.

- Ligne
- Triangle
- Texte
  - Label de l'edge
  - Rectangle stockant le texte
- **Background** : lignes horizontales et verticales
- **Origin point** : Rond rouge à (0,0)
- **Résultat recherche Dijkstra** : même dessin que l'edge, couleur violette avec le résultat comme label.

Les fonctions qui permettent de dessiner les différents éléments du Graph sont rassemblées dans la classe **Draw()**. Comme vu précédemment, le contrôleur l'appelle dans sa fonction de mise à jour de la vue **paintComponent()**

```

1      draw.grid();
2      draw.origin();
3      draw.all_arrows(nodes);
4      draw.nodes(nodes);
5      draw.node_circle(hoveredNodes);
6      draw.node_circle(clickednodes);
7      if (result != null) {
8          draw.path(result, nodes);
9      }

```

Voici ci-dessous la fonction de dessin des nœuds, les autres fonctions fonctionnent de manière analogue

```

1 public void nodes(HashMap<Integer, Node> nodes) {
2     for (Node node : nodes.values()) {
3         int x1 = Math.round(node.getPosition()[0]);
4         int y1 = Math.round(node.getPosition()[1]);
5         g2d.setColor(new Color(0, 100, 0));
6         g2d.fillOval(x1 - 5, y1 - 5, 10, 10);
7         g2d.setColor(Color.BLACK);
8         g2d.drawString(Integer.toString(node.getId()), x1 + 10, y1);
9     }
10 }

```

La fonction de dessin des *edges* est plus complexe, puisqu'elle se compose de quatre sous-éléments graphiques détaillés plus hauts.

```

1
2 public void all_arrows(HashMap<Integer, Node> nodes) {
3     for (Node node : nodes.values()) {
4         int x1 = Math.round(node.getPosition()[0]);
5         int y1 = Math.round(node.getPosition()[1]);
6
7         for (Integer neighborId : node.getEdges().keySet()) {
8             Node neighborNode = nodes.get(neighborId);
9             if (neighborNode != null) {
10                 int x2 = Math.round(neighborNode.getPosition()[0]);
11                 int y2 = Math.round(neighborNode.getPosition()[1]);
12                 arrow(x1, x2, y1, y2);
13                 texte_rectangle(x1, y1, x2, y2, node.getEdges().get(
neighborId).getLabel(), 3, "orange");
14             }
15         }
16     }
17 }

```

La plus grande difficulté aura été de générer les flèches pour indiquer la direction de la connexion, **nécessitant l'utilisation de calculs trigonométriques** pour trouver l'angle de

rotation, ainsi que l'ajustement dynamique de la boîte affichant le label en fonction de la longueur du texte.

```
1
2 public void texte_rectangle(int x1, int y1, int x2, int y2, String text, int
   position, String colorname) {
3     Color color = new Color(241, 97, 8, 190);
4     Color border = new Color(147, 60, 10, 255);
5     if (Objects.equals(colorname, "purple")) {
6         color = new Color(152, 70, 255, 194);
7         border = new Color(81, 45, 110, 255);
8     } else if (Objects.equals(colorname, "orange")) {
9         color = new Color(241, 97, 8, 190);
10        border = new Color(147, 60, 10, 255);
11    }
12    // draw string with distance
13    g2d.setColor(new Color(0, 0, 0, 255));
14    g2d.setFont(new Font("Helvetica", Font.BOLD, 14));
15    FontMetrics fontMetrics = g2d.getFontMetrics();
16    int textWidth = fontMetrics.stringWidth(text);
17
18    // Calculate the dimensions and position of the rectangle
19    int rectWidth = textWidth + 20; // Add some padding
20    int rectHeight = 35;
21    int rectX = x1 + (x2 - x1 - rectWidth) / position;
22    int rectY = y1 + (y2 - y1 - rectHeight) / position - 10;
23
24    // Draw the rounded rectangle
25    g2d.setColor(color);
26    g2d.fillRoundRect(rectX, rectY, rectWidth, rectHeight, 20, 20);
27    // add a white border
28    g2d.setColor(border);
29    g2d.drawRoundRect(rectX, rectY, rectWidth, rectHeight, 20, 20);
30    // Draw the text centered within the rectangle
31    g2d.setColor(Color.BLACK);
32    int textX = rectX + (rectWidth - textWidth) / 2;
33    int textY = rectY + (rectHeight - fontMetrics.getHeight()) / 2 +
fontMetrics.getAscent();
34    g2d.drawString(text, textX, textY);
35    }
36
37
38    public void arrow(int x1, int x2, int y1, int y2) {
39
40        double angle = Math.atan2(y2 - y1, x2 - x1);
41
42        int nodeRadius = 5; // Adjust the radius of the node circle as needed
43
44        // Calculate the adjusted start and end points
45        int startX = x1 + (int) (Math.cos(angle) * nodeRadius);
46        int startY = y1 + (int) (Math.sin(angle) * nodeRadius);
47        int endX = x2 - (int) (Math.cos(angle) * nodeRadius);
48        int endY = y2 - (int) (Math.sin(angle) * nodeRadius);
49
50        // Draw line with padding between the start and end of node
51        g2d.setColor(Color.BLACK);
52        g2d.drawLine(startX, startY, endX, endY);
53        // Print the weight with a little padding (5 pixels)
54        // Draw arrow
55        int arrowSize = 10;
56        int arrowX1 = (int) (x2 - arrowSize * Math.cos(angle - Math.PI / 6));
```





## Conclusion

Nous avons pu implémenter toutes les fonctions demandées, le tout dans une interface claire et jolie avec d'excellent feedback utilisateur. De plus, nous avons pu nous rendre compte de l'utilité des diagrammes UML afin de définir les besoins et la structure du projet de façon optimale, en particulier quand nous sommes plusieurs pour être sûr de partir sur la même base avec du code compatible.