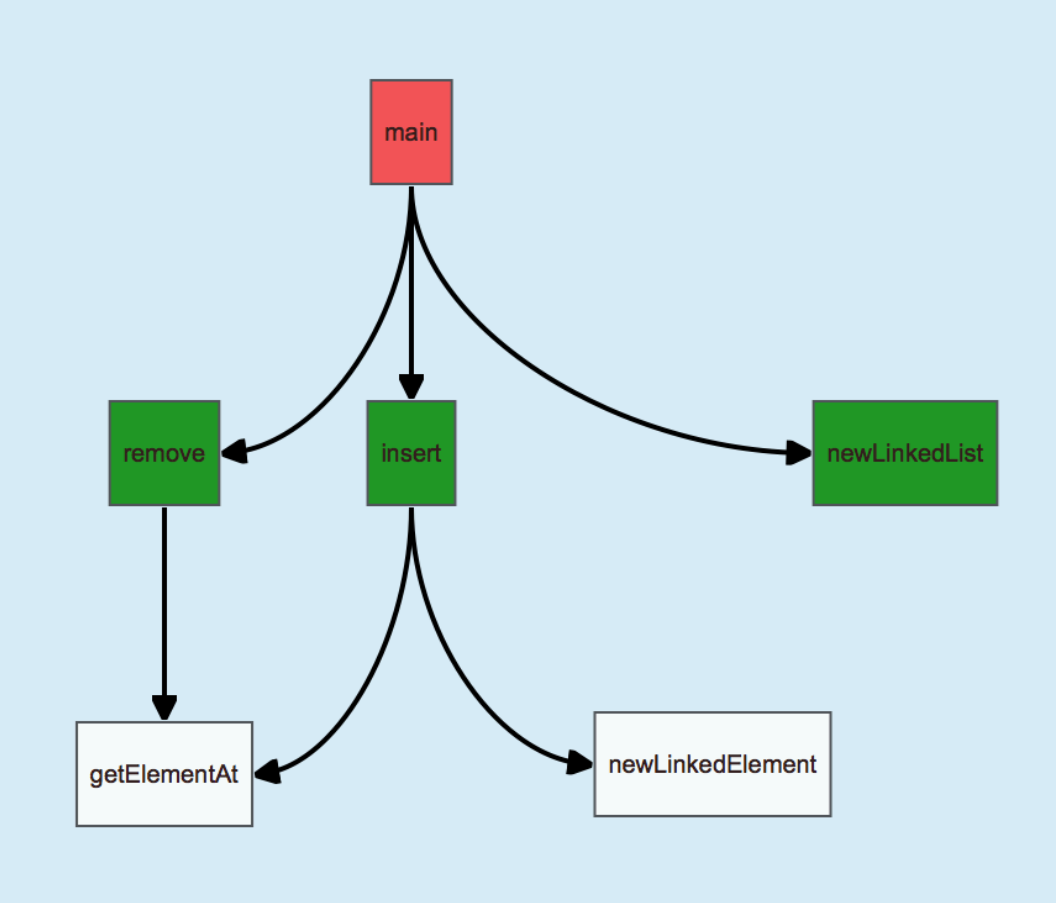
Projet 2012-2013

ISEN | CIR2

AlgoView Call Graph



Groupe 7:
DELMAIRE Matthieu
OLIVE Nicolas
DUFOUR Alexis

Référents:
SOULIGNAC Michaël
GAILLARD François

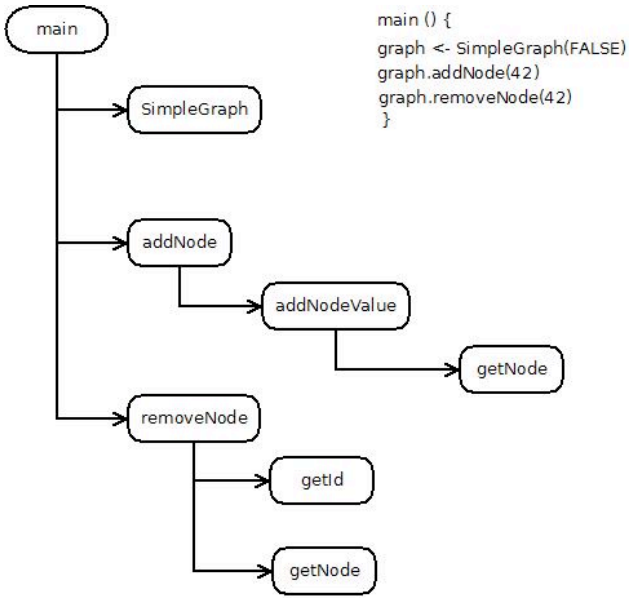
Table des matières

Introduction.

1. Objectif du projet.

Algoview est un compilateur/interpréteur de “Pseudo Langage” développé par Cédric Dinont et Michaël Soulignac. Dans un but pédagogique, il permet aux étudiants de mieux comprendre les concepts de base de la programmation.
Le projet informatique CIR2: 2012-2013 consiste à développer un plugin identique à celui présent dans Netbeans appelé “CallGraph”. Ce dernier permet de représenter graphiquement le graphe d’appel des fonctions d’un programme. Ce plugin permettra de représenter un graphe des appels de fonction dans un programme et il sera capable de repérer les cycles dus à des récursions.

Figure 0.1.1: Exemple d’un appel



Prenons le cas de la création d’un graphe:
•Le main appelle la fonction SimpleGraph avec des arguments.
•Ensuite le main appelle la fonction addNode qui elle même appelle des fonctions dont elle a besoin.
•Pour finir il appelle la fonction removeNode qui appellera aussi d’autres fonctions.

2. Technologie(s) utilisée(s).

Algoview étant codé en Javascript, le projet sera essentiellement développé dans ce langage pour être intégré par la suite dans ce dernier. Il respectera le patron MVC(Modèle Vue Contrôleur).
Afin de répondre au cahier des charges, notre application devra être développée grâce aux différentes technologies web vu au cours du premier semestre.
En effet, afin de mener à bien ce projet nous devrons notamment utiliser :

- Le javascript, qui servira d’une part pour la partie «interne» du projet(type graphe) et d’autre part dans la représentation graphique du graphe.
- HTML5 (HyperText Markup Language 5) est la dernière révision majeure d'HTML (format de données conçu pour représenter les pages web). Indispensable afin d’intégrer notre application dans la version JavaScript d’Algoview.

• SVG (Scalable Vector Graphics) est un format de dessin vectoriel, élaboré à partir de 1998 par un groupe de travail comprenant entre autre IBM, Apple, Microsoft, Xerox. Malheureusement cette technologie a connu un lente intégration dans les navigateurs.Explorer ne l'a pris en charge qu'a partir de sa version 9. Pour ce qui est des autres acteurs majeurs,SVG dans sa version 1.1 est reconnu et interprété, à partir de Firefox 4+, Chrome 16+, Safari 5+ et Opera 9.5+ .
Pourtant le svg présente de nombreux avantages :
SVG est un format d'image léger lorsqu'il s'agit de représenter des formes simples, car seules les informations décrivant ces formes sont stockées (coordonnées, couleurs, effets) contrairement aux images bitmap (JPG, PNG, GIF) qui doivent mémoriser le contenu pixel par pixel. Ce principe rend les images SVG étirables sans perte de qualité.

- ✳ Son apprentissage est facile car basé sur une syntaxe simple et intuitive
- ✳ Il peut être édité par un éditeur de texte basique car c'est du XML
- ✳ Il peut être manipulé via JavaScript, car présent dans le DOM
- ✳ Il peut être stylé grâce à CSS

De plus, grâce à l'arrivée de l'HTML 5 il est également très facile de l'intégrer dans sa page web.

Figure 0.2.1: Différence entre une image de format SVG et les autres formats

3. Organisation du projet.

Notre groupe se compose de trois personnes.
Le projet dure 13 semaines et se divise en trois parties(Implementation du type graphe, création de l'interface graphique, Integration du plugin) .
Le graphique ci-dessous montre le planning générale ainsi que les différents objectifs et leur date de rendu associée.

Figure 0.3.1: Organisation des séances

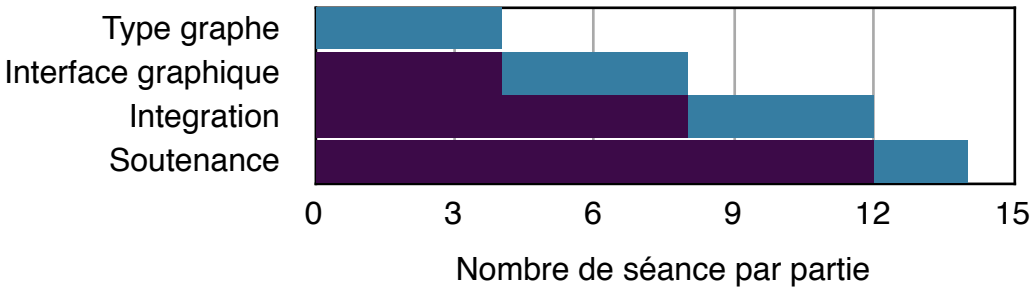


Figure 0.3.2: Livrables associés aux objectifs

Objectif		Livrables associés			
n°	Intitulé	Date Limite	Rapport	Code	Exigences devant être couvertes
1	Type Graphe	12/02/2013	x	x	1,2,3,4,6,7,8,9
2	Interface Graphique	26/03/2013	x	x	précédentes + 5, 10, 11, 12, 13
3	Intégration	21/05/2013	x	x	précédentes + 14, 15, 16

Un dépôt SVN est également à notre disposition pour que chacun puisse suivre l'avancement du projet.
Le rapport quand à lui est rédigé au fur et à mesure de notre avancement. Comme vu ci-dessus à chaque rendu d'un objectif, le compte rendu sera également demandé.

Objectif 1: Type Graphe

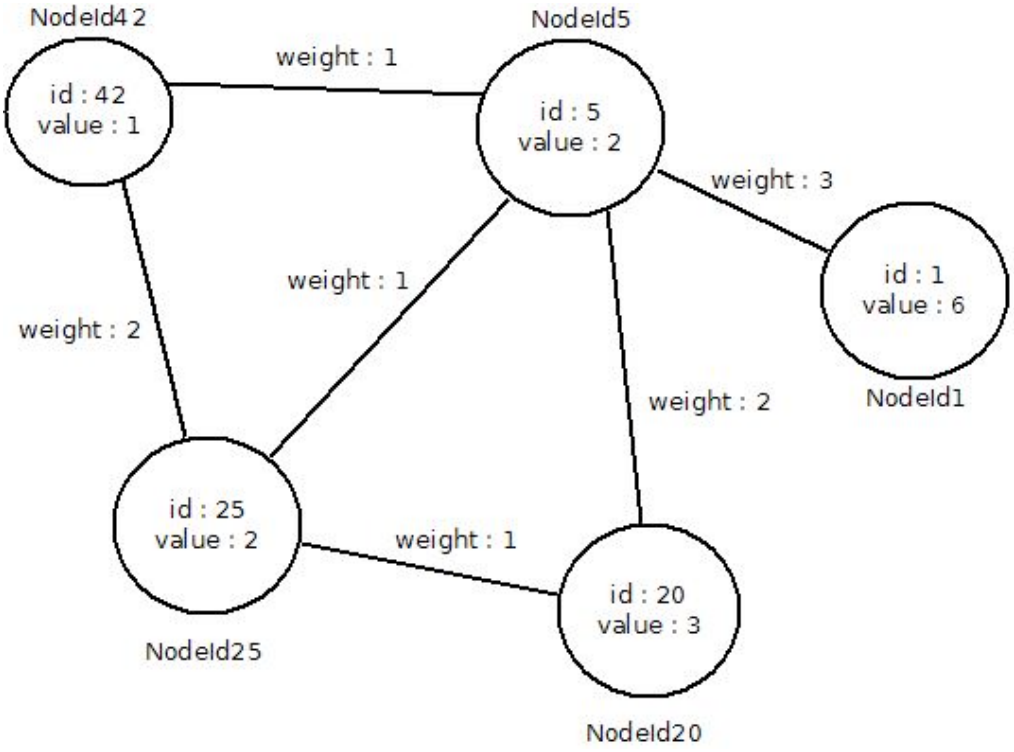
Le but principal de cet objectif est d'implémenter le type graphe étant l'implémentation la mieux adaptée à la réalisation de se projet.

4. Modélisation du type graphe.

- **Conception**

Notre choix s'est tout de suite porté sur le type Graphe vu en cours durant le premier semestre.Ce dernier permet de modéliser un ensemble d'éléments de même nature étant liés entre eux. Les éléments sont représentés par ce qu'on appelle "noeuds" du graphe. Les liaisons entre ces derniers sont quand à elles appelées "arcs".Voyons ci-dessous le fonctionnement global d'un graphe.

Figure I.4.1:Exemple de représentation d'un



Le graphe est en réalité l'ensemble des noeuds et leurs liaisons. Il est donc composé d'un ensemble de noeuds pouvant contenir des données ainsi que leur(s) relation(s) avec les autres, les relations pouvant contenir elles-mêmes des données.

Il est à noter qu'un graphe peut être dirigé ou non. C'est à dire que l'arc entre A → B n'est pas forcément le même qu'entre B → A.
Si le graphe est dirigé la création d'un arc entre le noeud A et le noeud B n'implique pas la création d'un arc entre B et A. À l'inverse, dans le cas d'un graphe non dirigé, la création d'un arc entre le noeud A et le noeud B implique également celle entre le noeud B et le A, les deux arcs étant un seul et même arc.

Lors des cours d'algorithmie nous avons pu découvrir deux façons d'implémenter le type Graphe :

- ✳ Grâce aux listes chainées
- ✳ Grâce aux matrices d'adjacences

Chacune d'elles ayant ses avantages et ses inconvénients.

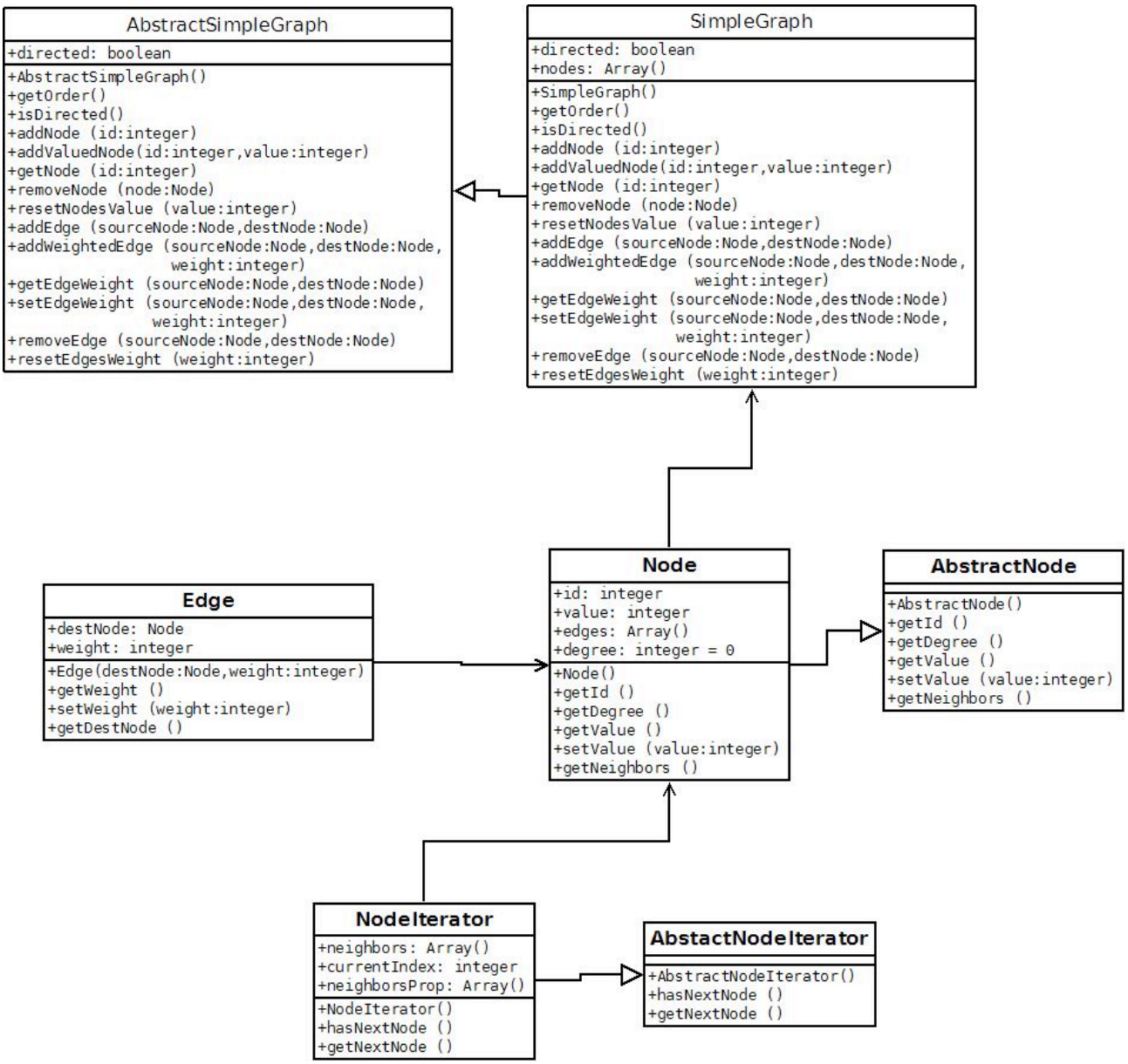
- **Choix d'implémentation**
  - ⋆ **Exigences**

    Avant de penser à l'implémentation en tant que telle du type Graph, exposons les exigences liées à l'objectif numéro 1 :
    - ✳ Notre implémentation du Graphe devra hériter de AbstractSimpleGraph, on devra également redéfinir et implémenter les fonctions vides fournies dans le squelette(*Ref : Figure I.1.2)*.
    - ✳ Notre implémentation des noeuds devra hériter de AbstractNode. Comme précédemment nous devrons redéfinir et implémenter les fonctions vides fournies dans le squelette.
    - ✳ Un itérateur sur noeud nous a également été demandé, il devra hériter de AbstractNodeIterator.
    - ✳ Enfin 6 exceptions sont à implémenter. (*Ref : Figure I.1.7)*

  - ★ **Graphe**

    Nous avons eu une première phase de réflexion et d'analyse du projet afin de choisir la meilleure implémentation possible.
    Premièrement nous nous sommes dits que le graphe d'appel des fonctions une fois crée n'était pas destiné à continuellement évoluer. En effet, en dehors de la phase de création du graph, les fonctions permettant d'ajouter ou supprimer des noeuds seront rarement appelées.
    En revanche l'accès aux données sera souvent sollicité.
    Les tableaux semblent être alors plus appropriés à ce projet.
    Nous décidons alors de diviser l'implémentation en 4 parties correspondant à 4 "classes" :
    - ✳ Graph
    - ✳ Node
    - ✳ Edge
    - ✳ NodeIterator

Figure I.4.2: Diagramme de classes du AbstractSimpleGraph:



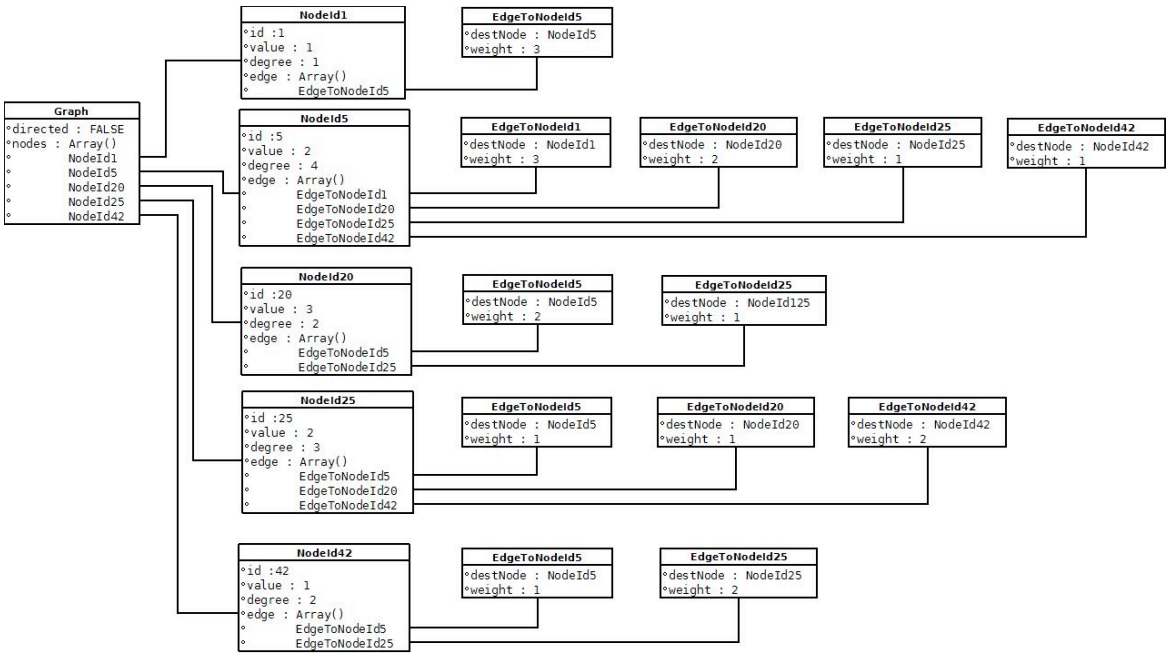
Il est à noter que le Javascript ne permet pas de créer des classes à proprement parler comme en C++ ou en PHP. Néanmoins il est possible de simuler ce comportement. De plus, le langage offre d'autres possibilités que les autres ne permettent pas, notamment le fait de pouvoir ajouter ou supprimer dynamiquement les propriétés d'un objet.
C'est d'ailleurs cette possibilité que nous avons exploité et sur laquelle repose notre implémentation.

En effet, nous avons décidé de représenter le graphe comme un objet ayant comme propriétés:

- ✴ Directed (Permet de savoir si le graphe est dirigé ou non).
- ✴ Nodes (Les noeuds qu'il contient)

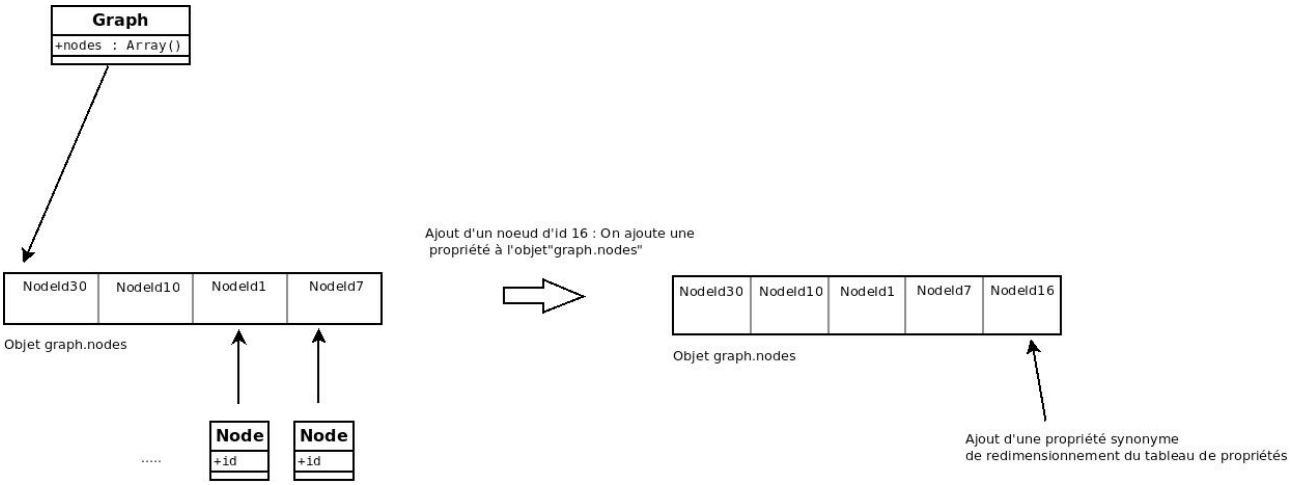
Les noeuds sont représentés comme des propriétés de l'objet SimpleGraph.Nodes. Le langage permet également de déléguer le travail de hachage ainsi que celui du redimensionnement du tableau de noeuds.
En effet, les propriétés d'un objet cachent en fait un tableau contenant ces dernières.
Les noeuds seront identifiés par une clé unique. Par convention leur clé sera du type «NodeId»+id du noeud.
De plus les performances des fonctions permettant l'ajout et la suppression de propriétés sont censées faire partie de celles qui sont le plus optimisées par les navigateurs étant donné que ce comportement des propriétés est une des bases du langage objet en javascript.
De la même manière les arcs seront des propriétés de l'objet Node.Edges. Tout comme les noeuds nous avons décidé d'une convention de nommage du type «EdgeTo»+id du noeud de destination
Ci-dessous un diagramme permettant de mieux comprendre les différents liens entre Graph, Node et Edge.

Figure I.4.3: Diagramme UML d'un graphe selon notre implémentation:



Ainsi que ci-dessous une explication en image de l'ajout et la suppression de propriétés dans un objet en s'appuyant sur l'exemple d'un ajout ou d'une suppression de noeud.

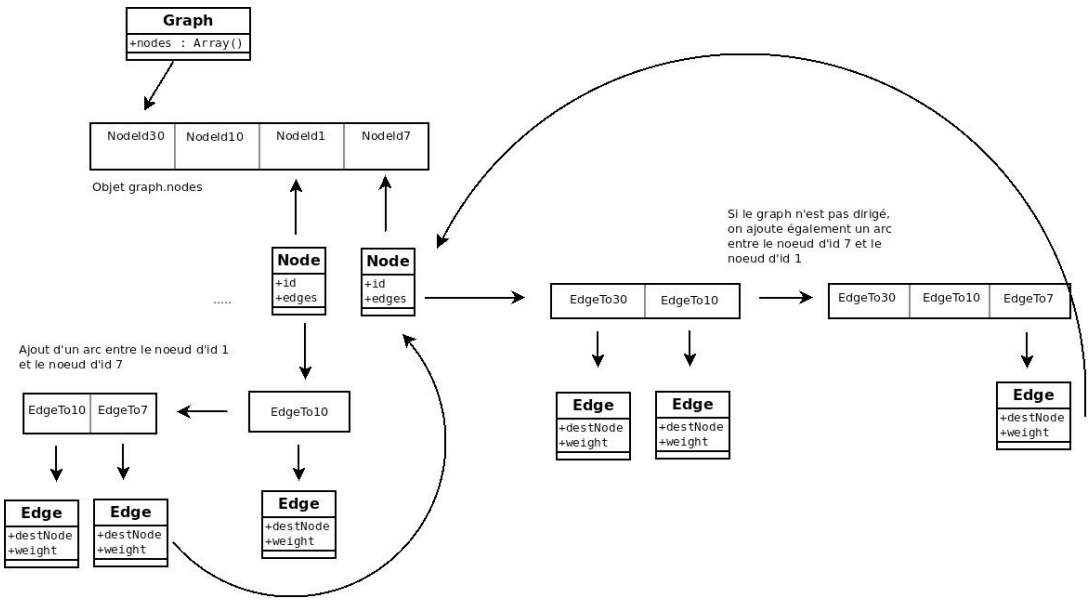
Figure I.4.4: Exemple d'ajout d'un noeud:



Ci-dessus on peut voir que l'objet Graph admet une propriété "nodes" elle même étant un objet elle admet des propriétés.
Ces propriétés sont en fait dans notre application les noeuds contenus dans le graphe. Lors de l'ajout d'un noeud on ne fait en fait qu'ajouter une propriété à l'objet Graph.Nodes, le travail du redimensionnement du tableau de propriétés de l'objet est délégué à Javascript.

On retrouve le même comportement dans l'objet Node:

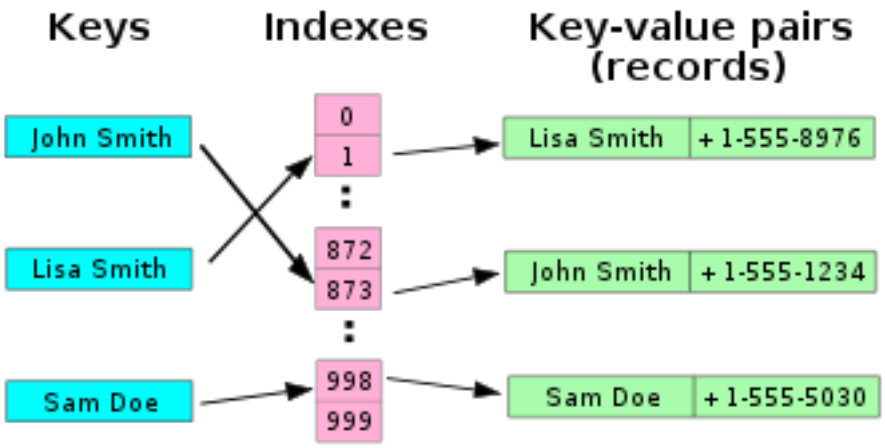
Figure I.4.5: Exemple d'ajout d'un arc entre



Les propriétés ne sont donc pas repérés dans le tableau par leur index mais par leur clé qui, rappelons-le est du type :
- «NodeId»+id du noeud
- «EdgeTo»+id du noeud destination

Rappelons au passage le principe du hachage :
On accède à chaque élément du tableau via sa clé. L'accès à un élément se fait en transformant la clé en une valeur de hachage par l'intermédiaire d'une fonction de hachage.
Ci-dessous une image illustrant le fonctionnement des tables de hachage.

Figure I.4.6: Fonction du table de hachage:

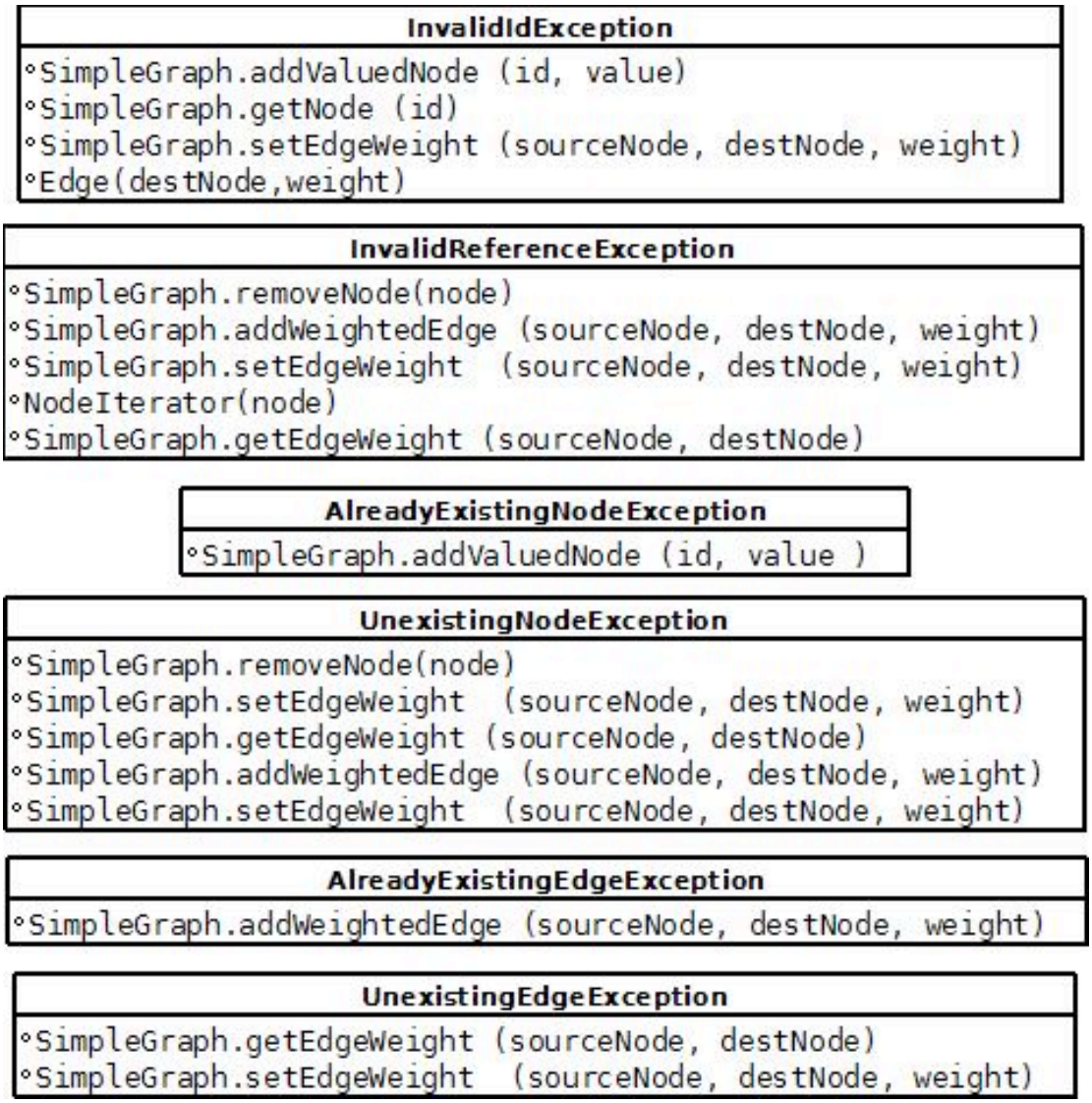


Néanmoins toute cette étape de hachage nous ai déjà fourni par le langage, nous n'avons donc pas à nous soucier des performances de cette dernière.

★ Exceptions

Dans le cadre de notre premier objectif nous devions également implémenter différents types d'exceptions afin de développer une application robuste aux erreurs et profitant des avantages du langage orienté objet.
Toutes les exceptions héritent du type GraphException. Le diagramme ci-dessous permet d'illustrer les différentes exceptions de notre application.

Figure I.4.7: Illustration des exceptions:



5. Tests et Performances:

- **Résultats des tests**

Dans ce projet la campagne de tests nous a été fourni. Elle comporte 488 tests effectués grâce à QUnit une API de tests unitaires en Javascript.
Cette campagne de tests porte sur l'ensemble du type Graphe(tests sur toutes les fonctions que nous devions implémenter), elle permet ainsi de savoir si notre implémentation respecte bien l'ensemble du cahier des charges.
Le résultat est plutôt satisfaisant étant donné que nous passons l'ensemble des tests avec succès sur Google Chrome comme sur Firefox.

Figure I.5.1: Résultats de nos tests:

- **Performance par rapport au choix d'implémentation.**

Intéressons nous désormais aux performances de nos fonctions. Ci-dessous vous pourrez trouver le des différentes fonctions que nous devions implémenter ainsi que leur complexité.

Figure I.5.2: Complexité de notre implémentation

Complexité Temporelle m : nombre d'arcs n : nombre de nœuds g : nombre d'arcs dans destNode			
Fonction	Complexité minimal	Complexité maximal	Complexité moyenne
addNode	Θ(n)	Θ(n)	Θ(n)
getNode	Θ(1)	Θ(1)	Θ(1)
resetNodesValue	Θ(n)	Θ(n)	Θ(n)
addEdge	Θ(m)	Θ(m)	Θ(m)
getEdgeWeight	Θ(1)	Θ(1)	Θ(1)
setEdgeWeight	Θ(1)	Θ(1)	Θ(1)
resetEdgesWeight	Θ(n)	Θ(m*n)	Θ(m*n)
removeEdge	Θ(m))+ Θ(g)	Θ(m)+ Θ(g)	Θ(m))+ Θ(g)
removeNode	Θ(n)	Θ(n*m)	Θ(n*m)

Nous avons également effectué des tests "concrets" afin de relever le temps nécessaire pour effectuer un certains nombre de fois chaque fonction.
Nos tests se décomposent donc en 3 parties :

- Tests sur un graphe de 100 noeuds
- Tests sur un graphe de 1000 noeuds
- Tests sur un graphe de 10 000 noeuds

Nous décidons également d'effectuer nos tests sur Chrome et FireFox.
Ci-dessous un tableau reprenant nos résultats finaux.

Figure I.5.3: Temps d'exécution de chaque fonction dans chrome et firefox:

Nombredenoeuds	100		1000		10000	
Navigateur	Chrome	Firefox	Chrome	Firefox	Chrome	Firefox
addNode	0ms	1ms	2ms	5ms	19ms	40ms
getNode	0ms	0ms	1ms	1ms	3ms	10ms
ResetNodesValue (1iteration)	0ms	0ms	1ms	0ms	2ms	2ms
addEdge	1ms	1ms	5ms	9ms	25ms	95ms
getEdgeWeight	0ms	0ms	2ms	6ms	25ms	70ms
setEdgeWeight	0ms	0ms	4ms	6ms	15ms	61ms
ResetEdgesWeight (1iteration)	0ms	0ms	2ms	0ms	6ms	10ms
RemoveEdge	0ms	1ms	4ms	6ms	36ms	67ms
RemoveNodeavecedge	4ms	4ms	224ms	340ms	30083ms	35693ms
RemoveNodesansedge	3ms	4ms	243ms	343ms	29406ms	35270ms

On peut voir dans ce tableau que les opérations les plus coûteuses sont celles d'ajout et de suppression. Comme l'avait laissé paraître l'étude des complexités.

6. Conclusion

Notre choix d'implémentation s'est donc tourné vers les tableaux ces derniers permettant un accès constant aux données. De plus, nous avons décidé de nous appuyer sur les spécificités de JavaScript notamment l'ajout et la suppression dynamique de propriétés. Cette dernière nous permet alors de nous délester d'une partie du travail comme le hachage des tableaux de propriétés ou encore leur redimensionnement.
Notre implémentation répond aux cahier des charges comme le montre les résultats de nos tests unitaires.
Au niveau des performances les fonctions concernant l'ajout ou la suppression de noeud ou d'arc sont celles qui sont les plus "gourmandes". Néanmoins, ces dernières seront, comme nous l'avons déjà dit, moins utilisées.

Objectif 2: Interface Graphique

L'objectif principal sera d'implémenter une vue graphique en utilisant la technologie SVG et un vue texte. Celles-ci nous permettront de modéliser un graphe de différente manière.

7. Présentation des bibliothèques et des technologie utilisées

- **RaphaëlJs**

Pour notre première vue graphique nous avons décidé d'utiliser la librairie RaphaëlJs . C'est une librairie JavaScript permettant de faire du dessin vectoriel, en utilisant pour cela, SVG et VML.
RaphaëlJS est supportée par la quasi totalité des navigateurs (Safari 3.0+, Opera 9.5+, Firefox 3.0+, Chrome 5.0+ et même Internet Explorer 6.0+).
Avec cette librairie JavaScript, tout les éléments graphiques que nous allons créer seront des éléments du DOM.
Nous pouvons donc les manipuler comme nous le souhaitons. De plus le fait que cette bibliothèque utilise la technologie SVG offre de nombreux avantages que nous détaillons ci-dessous.

L'un des gros avantages de Raphael est qu'il facilite énormément la création de forme en SVG, notamment grâce à ses multiples fonctions disponibles(rectangle, cercle, ellipse...).
Dans l'ensemble RaphaelJs est une bibliothèque accessible à tous, néanmoins il faut parfois s'attarder quelques temps sur le code pour comprendre comment tout cela se passe (elle fait parfois appel aux mathématiques pour la construction de certaines formes), en plus du temps pour découvrir les multiples fonctions qu'offre cette dernière. Le défi ici était de pouvoir adapter la bibliothèque à notre objectif. Après du temps passé à se documenter et à l'analyser, Raphael se révèle être une bibliothèque très puissante permettant un gain de temps et de travail considérable ainsi qu'une portabilité assez impressionnante.

- **Le VML et le SVG**

Le VML (Vector Markup Language) est un langage XML, destiné à la création des graphismes vectorielles élaborés en 2D ou 3D (statiques ou animées) sur les pages Web. Ce dernier sera combiné avec un autre format : le PGML(Precision Graphics Markup Language) qui donneront naissance au SVG(Scalable Vector Graphic).
Ce format est rappelons le, conçu pour décrire des ensembles de graphiques vectoriels. Il gère la plupart des formes connues tels que les rectangles, cercles , ellipses etc.. Mais également les chemins et courbes de Béziers. Tout ceci en fait un outil très puissant pour pouvoir dessiner n'importe quelle forme.

- **jQuery**

jQuery est une bibliothèque JavaScript qui porte sur l'interaction entre le JavaScript et HTML, elle a pour but de simplifier des commandes communes de JavaScript. La plupart des bibliothèques que nous avons utilisé ici, l'utilise. Ceci montre qu'elle est devenue rapidement incontournable dans le monde du WEB, notamment grâce au fait qu'elle soit très portative et permet ainsi d'avoir une même syntaxe pour tous les navigateurs.
Dans notre application cette bibliothèque nous sert essentiellement à parcourir le DOM ou à ajouter ce que l'on appelle des écouteurs. Ces derniers permettent d'appeler une fonction lors d'une interaction de l'utilisateur (clic de la souris, pression d'une touche du

clavier...). C'est eux qui préviennent le contrôleur qu'il s'est passé quelque chose. Ce dernier peut donc réagir en conséquence.

- **dTree**

dTree est une bibliothèque Javascript permettant de créer facilement une arborescence de type "explorateur de fichier".
Son principe est relativement simple, on a un objet «treeNode» qui initialise un "noeud" ayant pour argument: un id, l'id de son parent, son nom ainsi que la possibilité de définir l'icône désirée, d'autres arguments sont présent dont nous n'aurons pas obligatoirement besoin.
Le noeud root (qui est à la base de l'arborescence) doit avoir l'id «-1», ceci étant une convention fixé par la bibliothèque. Ensuite on ajoute simple des noeuds grâce à la méthode «add» de dTree, lorsque cela est terminé il ne reste plus qu'à appeler la méthode «toString» qui génère alors le codeHTML permettant d'avoir la vue graphique de l'arborescence précédemment établie.

Figure II.7.1: Les différents paramètres permettant de créer un «treeNode»:

Paramètres		
Nom:	Type:	Description:
id	Number	Identifiant du noeud (Unique)
pid	Number	Identifiant du noeud parent
name	String	Le nom du noeud(Celui qu'on retrouvera à l'affichage de l'arborescence)
url	String	Url du noeud(Si contient un lien vers quelque chose) (facultatif)
title	String	Donne un titre au noeud (facultatif)
target	String	Une «cible» pour le noeud (facultatif)
icon	String	une icône quand le noeud est fermé
iconOpen	String	une icône quand le noeud est ouvert
open	Boolean	«True» si le noeud est ouvert sinon «False» (facultatif: Il sera à «False» par défaut)

Il est à noter que pour l'initialisation d'un objet "dTree" il faut veiller à lui passer en argument le même nom que la variable dans laquelle on stocke cet objet.
Exemple d'initialisation: var myVar = new dTree("myVar");

En effet le constructeur de dTree prenant comme argument le nom de l'objet qu'il va générer. On peut donc rapidement avoir des surprises si l'on ne respecte pas cette règle.

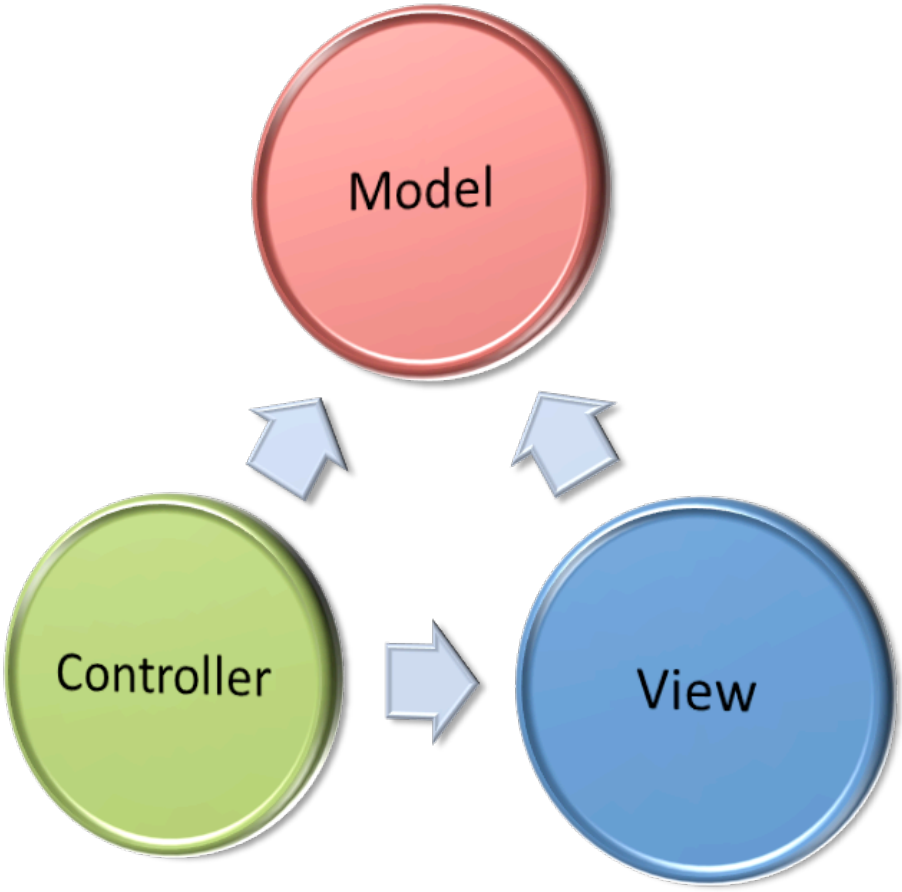
8. Introduction au MVC(Modèle Vue Contrôleur)

- **Les bases du MVC**

  En 1978-79 Trygve Reenskaug effectue des travaux ayant pour but principal de proposer une solution générale aux problèmes d'utilisateurs manipulant des données volumineuses et complexes.
  Le MVC (modèle vue contrôleur) aboutira de ces derniers.
  Il est destiné à répondre aux besoins des applications interactives en séparant les différentes problématiques de chaque partie de l'application.
  On distingue les 3 parties suivantes :

  - Le modèle (modèle de données)
  - Le(s) vue(s) (présentation, interface utilisateur)
  - Le contrôleur (logique de contrôle, gestion des événements, synchronisation)

Figure II.8.1: Représente la conception du MVC



Expliquons un peu plus en détail le rôle de chaque catégories de ce patron de conception.

★ **Le modèle**

Le *modèle* représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application. Il regroupe la gestion de ces données et est responsable de leur intégrité. Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données. Les résultats renvoyés par le modèle ne s'occupent pas de la présentation.

★ **La vue**

Ce avec quoi l'utilisateur interagit se nomme précisément la *vue*. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toute action de l'utilisateur (*hover*, clic de souris, sélection d'un bouton radio, cochage d'une case, entrée de texte, de mouvements, de voix, etc.). Ces différents événements sont envoyés au contrôleur. La vue n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

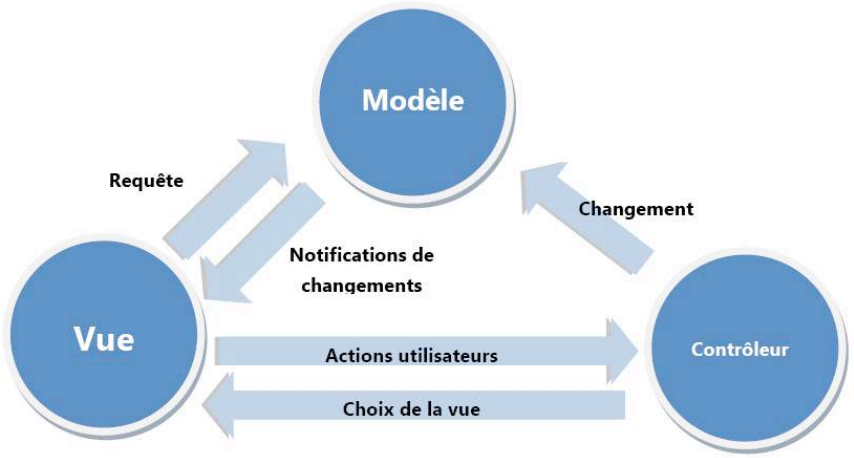
★ **Le contrôleur**

Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle, et ce dernier notifie la vue que les données ont changé pour qu'elle se mette à jour.

• **Fonctionnement du MVC**

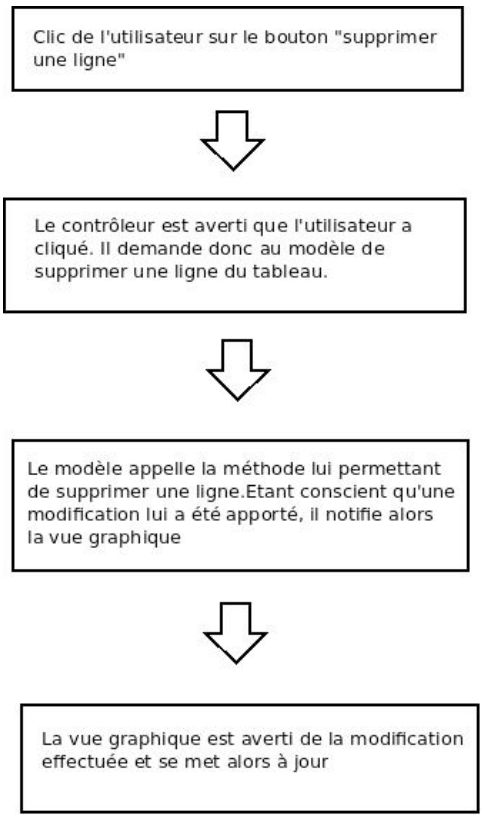
Maintenant intéressons nous au fonctionnement « interne » du MVC, plus précisément comment les catégories communiquent entres elles.
Commençons par ce schéma expliquant les interactions entre le modèle, la vue et le contrôleur.

Figure II.8.2: Schéma représentant les interactions entre le modèle, la vue et le contrôleur



Prenons un exemple simple afin de mieux détailler. Imaginons dans une application un bouton permettant de supprimer la ligne d'un tableau:
on aurait ici le modèle qui serait la représentation des données du tableau en mémoire.
La vue correspondrait à la représentation graphique du tableau.
Enfin le contrôleur lui serait alerté d'une modification grâce au bouton de suppression.

Figure II.8.3 :Le schéma ci-dessous représente les différentes étapes qui s'opèrent lors d'un clic sur le bouton de suppression d'une ligne.



9. Conception et implémentation

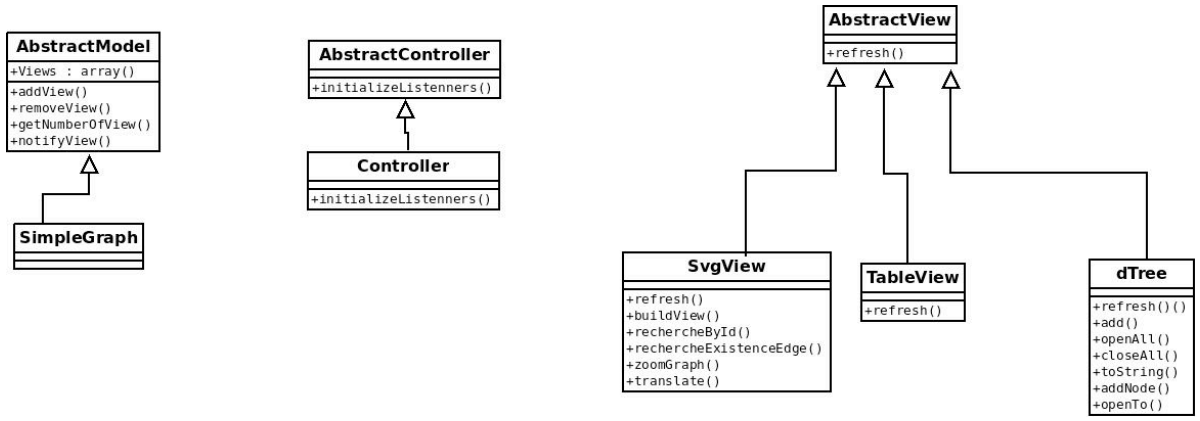
Afin de répondre aux exigences de l'objectif n°2 notre application doit respecter le patron MVC. Ainsi que posséder au minimum 2 vues :

- Une vue graphique réalisée en SVG
- Une vue texte présentant l'id du nœud ainsi que ses voisins

Étant le seul groupe de trois personnes, nous décidons de réaliser une vue supplémentaire qui consistera en une arborescence présentant les relations entre les différents nœuds.

Les choix de conception qui s'en suivent sont la création de 3 nouvelles classes abstraites (AbstractModel, AbstractView et AbstractController). De ces classes sont héritées les éléments que nous utilisons (vue texte, vue graphique, modèle de jeu ou contrôleur). Voici ci-dessous la représentation de nos choix d'implémentation afin de répondre aux différentes exigences de l'objectif n°2 et plus particulièrement celles relatives au MVC.

Figure II.9.1 : Schéma représentant le MVC de notre application :



- **Point de vue MVC**
  - ⋆ **Notre modèle**

    Notre modèle correspond ici à ce que nous avons du réaliser lors du premier objectif, c'est à dire le « Type Graph ». C'est lui qui gère la représentation de l'application en mémoire (ajout ou suppression de nœuds,d'arcs etc...).
    Néanmoins afin qu'il respecte au mieux nos objectifs, nous avons du le faire hériter de notre classe « AbstractModel » afin qu'il hérite des méthodes :addView, removeView, notifyView ainsi que getNumberOfView permettant respectivement d'ajouter/de supprimer une vue, de toutes les notifier d'un changement ou encore de récupérer le nombre de vue associées au modèle.

  - ⋆ **Nos vues**

    Toute nos vues héritent de la classe abstraite AbastractView afin d'hériter de la méthode refresh que nous avons surchargé pour chaque vue.
    Ces dernières ayant bien entendu des méthodes de construction et de mise à jour différentes.

  - ⋆ **Notre contrôleur**

    Lors de sa création ce dernier prend un modèle en argument. C'est le principe même du MVC le contrôleur connaît son modèle.
    Ainsi dans le "constructeur" de "Controller" on appelle sa méthode initializeListenners qui vient ajouter des écouteurs sur nos différentes vues afin qu'elles puissent toutes interagir entre elles.
    Lors d'un clic le contrôleur est alors prévenu étant donné que les écouteurs ont été initialisé à l'intérieur d'une de ses méthodes, les fonctions réagissant au clic appelleront alors des méthodes du modèle, qui elles même avertiront les vues de la modification apportée.

- **Vue graphique**

  Pour la représentation graphique, nous avons décidé d'utiliser Raphael. Ce dernier est une librairie javascript permettant une création facile d'un graphe, utilisant et simplifiant l'utilisation du SVG.
  De plus des écouteurs sont déjà implémentés au sein de la bibliothèque .

⋆ Principe de la création de la vue graphique

✣ Création des noeuds

On définit une zone graphique permettant de dessiner sur la page html. Chaque noeud du graphe est positionné en fonction de sa profondeur, du nombre de noeud à l'étage courant et de la profondeur maximale du graphe.
Pour cela on prend la taille de la zone graphique sur la longueur et on le divise par la profondeur maximale du graphe plus un. Ainsi on définit des paliers suivant la profondeur du noeud. Pour le positionnement de celui-ci sur le palier, on effectue le même procédé que pour les paliers en profondeur mais cette fois, avec le nombre de noeuds sur l'étage courant du noeud.

Le champ texte du noeud peut aussi influencer le positionnement de ce dernier sur la vue graphique. Nous testons si la taille du texte ne dépasse pas celle d'un palier suivant la largeur . Si c'est le cas, nous avons mis en place plusieurs protocole. Le premier étant d'écrire le texte sur deux lignes et non plus sur une ligne, de cela on réduit la taille du champ texte par 2.

Figure II.9.1 :

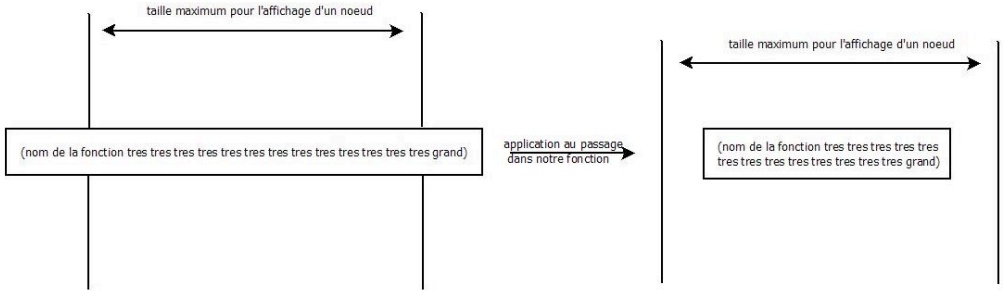


Schéma représentant le traitement d'un champ texte d'un noeud lorsqu'il est trop grand

Si cela ne suffit toujours pas, alors on applique le deuxième protocole qui est de déplacer la position du noeud soit au dessus ou en dessous du palier sur lequel il était placé.
Le fait de faire cela permet aussi d'éviter la collision entre deux noeuds ayant un champ texte très grand.

Figure II.9.2 :

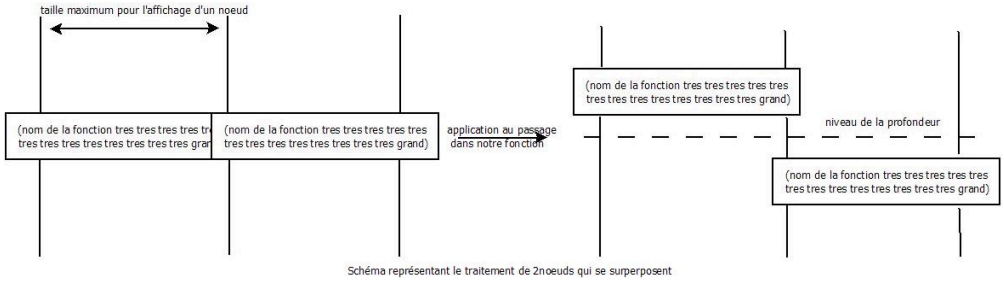
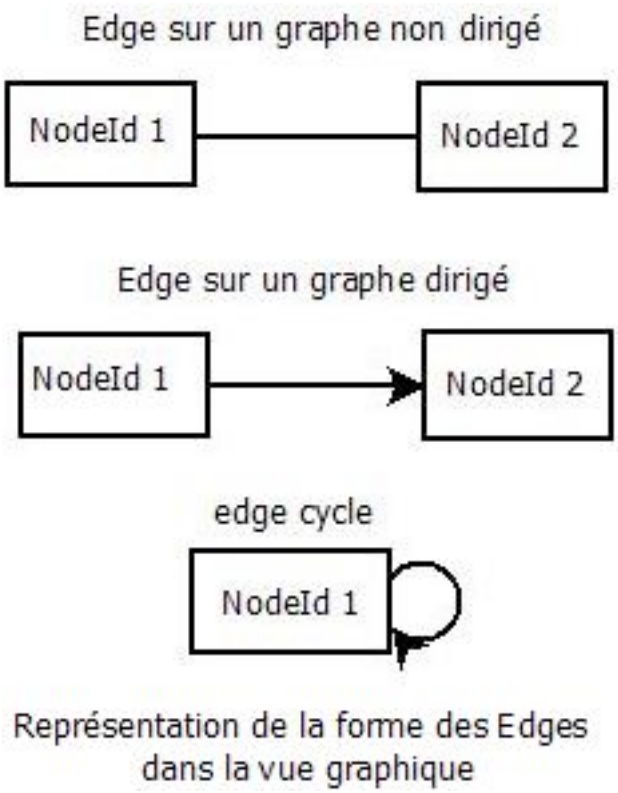


Schéma représentant le traitement de 2noeuds qui se surperposent

- **Création des arcs**

Pour la création des «Edges», on est venu ajouté une fonction dans la librairie Raphael.js: Raphael.fn.connection qui prend en compte si le graphe est dirigé ou non ou encore si l'edge est un cycle (Noeud ayant un lien sur lui même). Cette fonction prend en paramètre : deux noeuds, la couleur pour l'edge et le graphe. Dans la fonction, on créé la représentation de l'edge en SVG a partir des fonctions de Raphael sous forme d'une courbe. Ceci permet d'éviter la superposition de l'edge et des noeuds déjà existants. Si le graphe est dirigé, on crée un arc sous forme de flèche.

Figure II.9.3 :



Représentation de la forme des Edges dans la vue graphique

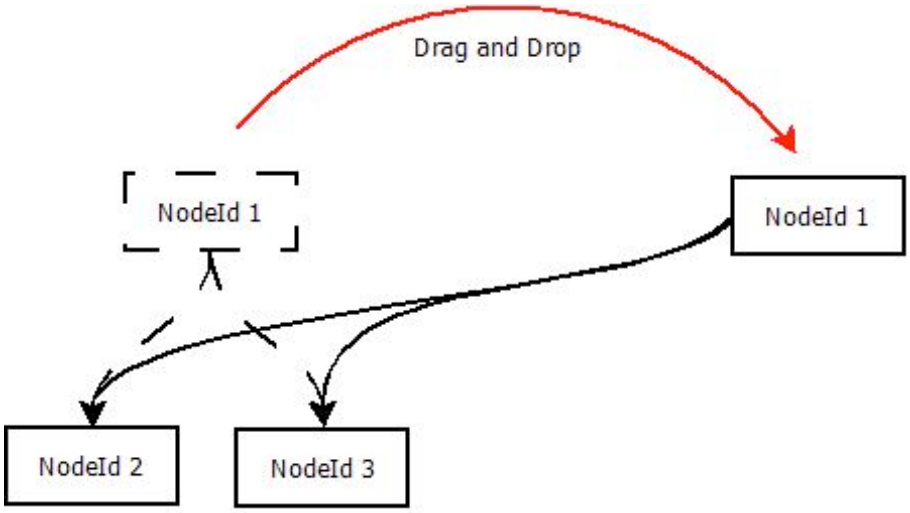
★ **Évènement**

Dans la vue graphique les fonctions suivantes ont été implémentées: déplacement des noeuds, translation du graphe et le zoom.

- **Déplacement d'un noeud**

Pour le déplacement des noeuds, on définit deux fonctions dans la création de la vue graphique, start et move.
Start permettant d'initialiser le move, on sauvegarde sa position initiale et dans la fonction move, on déplace le noeud là où l'utilisateur le souhaite en ajoutant à sa position initiale le décalage suivant: x,y du au déplacement du noeud. De plus tout les edges en relation avec le noeud bougent avec le noeud sélectionné.

Figure II.9.4 :



Représentation d'un déplacement d'un noeud,
les edges suivent bien le déplacement du noeud

- **Zoom**

Les fonctions de translate et de zoom reposent toutes deux sur la même utilisation de la fonction setViewBox proposé par Raphael. Le principe de setViewBox est de redéfinir la vue sur la zone graphique en spécifiant de nouvelles frontières . On ajoute des écouteurs sur la balise accueillant la zone de graphique, click, mouseup, mousemove et mousedown. L'évènement click est utilisé pour le zoom. L'utilisateur doit exécuter une combinaison pour activer ce dernier (Ctrl+click pour un zoom grossissant et Shift+click pour un zoom rétrécissant). Lorsque l'évènement est activé, on récupère l'endroit où l'utilisateur veut agrandir/diminuer la vue graphique et on appelle la fonction setViewBox avec des nouvelles frontières définit suivant le zoom tout en lui passant les coordonnées du clique en arguments.

Figure II.9.5 :

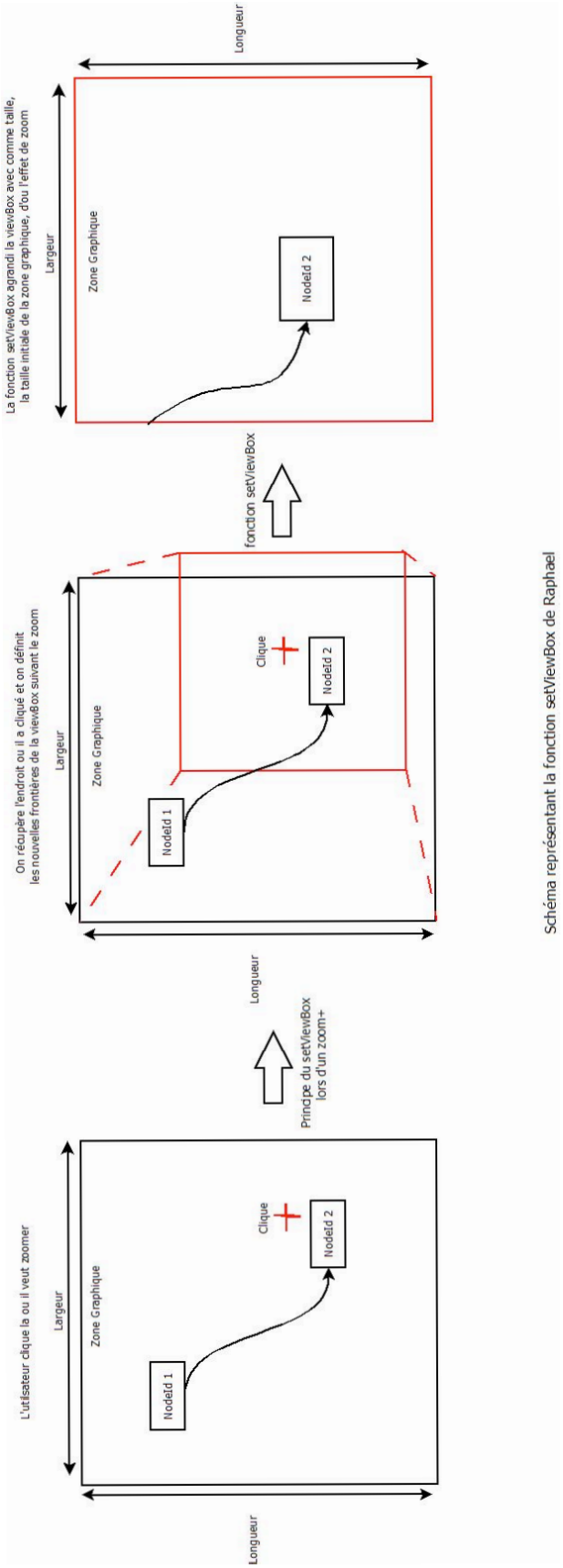
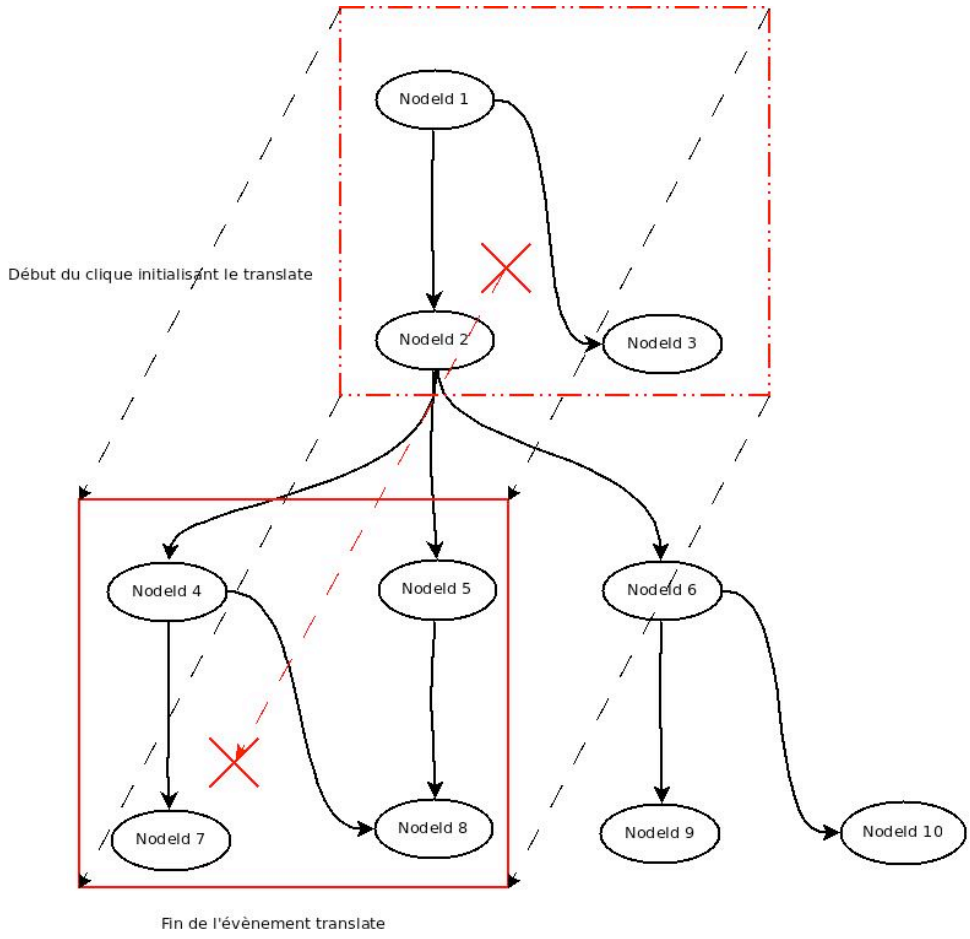


Schéma représentant la fonction setViewBox de Raphael

✣ **Translate**

Les événements mouseup, mousemove et mousedown sont utilisés pour la fonction translate. Lorsque l'utilisateur clique sur la zone graphique puis déplace sa souris, on déplace le graphe suivant sa direction. Pour cela on utilise setViewBox avec les mêmes frontières mais avec des coordonnées de viewBox différentes. Puis quand l'utilisateur relâche son clique, on arrête le déplacement de la viewBox.

Figure II.9.6: Principe de fonctionnement du «translate»



★ **Campagne de test**

Lors de la campagne de test, nous avons décidé de tester des cas de création de graphe basique(non dirigé, dirigé et cyclique) mais aussi des cas ou on teste les limites de notre code.

✳ Test sur le nombre de noeuds qu'on peut mettre sur une profondeur. On observe que notre code permet l'affichage d'un grand nombre de noeuds sur la profondeur grâce aux tests lors de la création des noeuds sur la vue graphique mais celle ci a quand même des limites.

L'affichage de 31 noeuds ayant un petit champ texte est représenté correctement mais l'affichage de 61 noeuds ayant un petit champ texte est représenté avec des défauts. Début de superposition entre les noeuds mais aussi des représentations

des edges qui passent par dessus les noeuds.

✳ Test sur l'influence du champ texte d'un noeud sur la représentation graphique du graphe. Lorsque le champ texte d'un noeud devient trop grand, on lui applique bien un retour a la ligne et si cela ne suffit pas, on applique alors le décalage sur la représentation du noeud suivant l'axe des ordonnées par rapport au niveau de la profondeur. Comme pour le test sur le nombre de noeuds qu'on peut mettre sur une profondeur, on y retrouve la même limite, à partir d'un certain nombres de noeuds ayant un champ texte assez grand, on observe une superposition des noeuds entre eux et aussi avec les edges.

Figure II.9.6 :

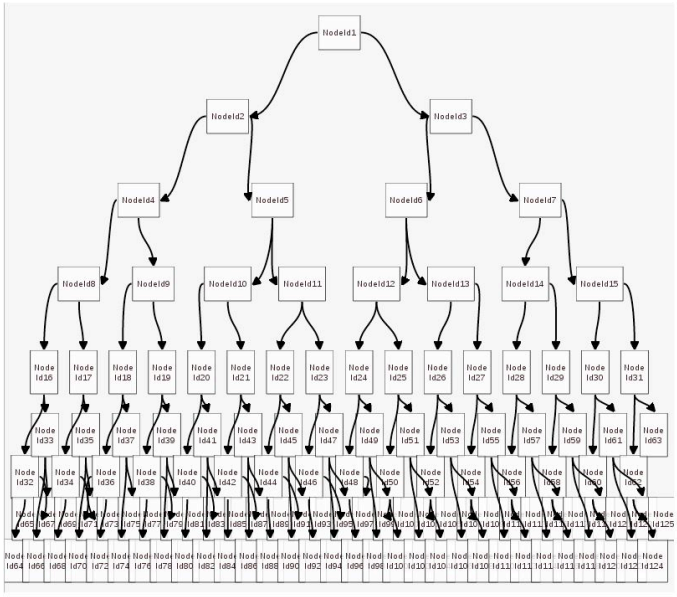


Schéma représentant les limites de l'affichage de notre vue graphique
sans modification de la part de l'utilisateur

- **Vue texte**

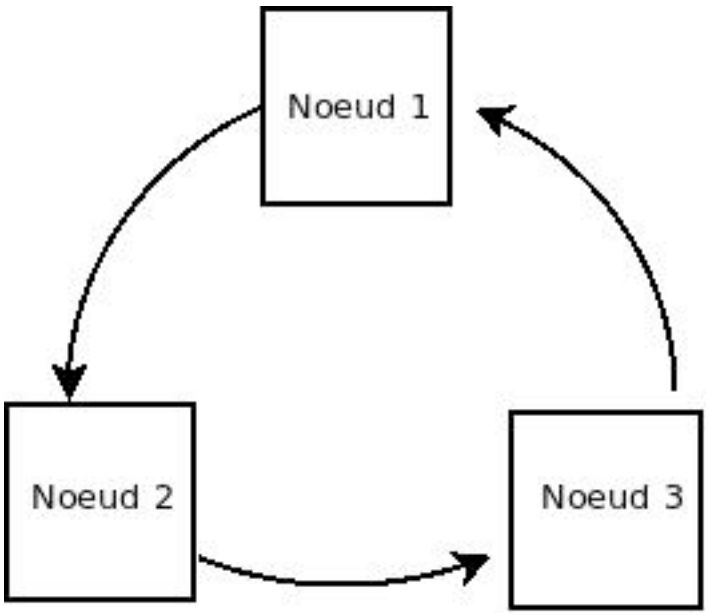
Cette vue permet à l'utilisateur d'avoir sous forme d'un tableau : l'id, la valeur et les voisins (désignés par leur id) du nœud couramment sélectionné.
Cette dernière se met alors à jour lors de la sélection d'un nœud dans l'une des deux autres vues de l'application.

- **Vue type arborescence de fichier**

Cette vue quand à elle permet à l'utilisateur de voir plus précisément les dépendances entre les différents nœuds et de pouvoir naviguer entre ces-derniers sous une forme différente.
Afin de la réaliser nous optons pour l'utilisation de la bibliothèque « dTree » présentée ci-dessus.
Néanmoins il a quand même fallu effectuer quelques modifications sur cette dernière afin d'obtenir le résultat attendu.
Ces modifications sont essentiellement des ajouts de nouvelles méthodes :
Comme toutes les autres vues on est venu surcharger la création de la vue ainsi que son rafraîchissement.
Pour la création de cette vue on effectue un parcours en largeur, afin de construire au

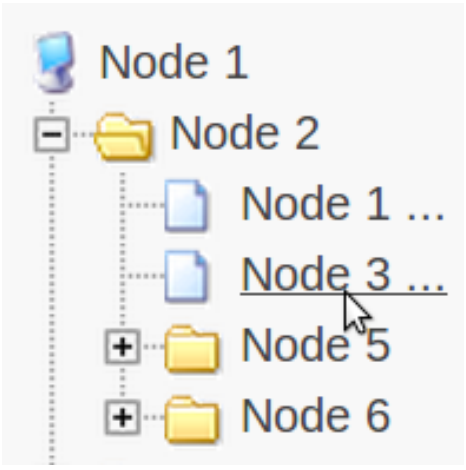
fur et a mesure l'arbre à l'aide des méthodes déjà fournis par la bibliothèque (ajout / suppression de nœud).
La seule difficulté rencontrée lors de cette surcharge de méthode était de détecter les cycles.
En effet, imaginons 3 nœuds : nœud 1, nœud 2, nœud 3. Ci-dessous les relations qui existent entre eux.

Figure II.9.7 : Ci-dessous une image représentant un cycle simple



Si l'on tombe sur cette configuration dans un graphe sans s'être soucié de la détection des cycles on risque d'obtenir une erreur. En effet la vue arborescente créera un nœud 1 contenant un nœud 2 qui lui même contient un nœud 3. Mais dans le nœud 3 se trouvera le nœud 1 et c'est la boucle qui recommence.
Tout comme lorsque un nœud aura un lien vers lui même la vue le détectera et l'affichera de la même manière que sur l'image ci-dessous

Figure II.9.8: Représentation d'un cycle dans la vue arborescente:



Ici on a le noeud 1 qui a pour voisin le noeud 2. Ce dernier a pour voisin le noeud 1, on est donc dans une configuration de récursion:
En effet si on parcourt le graphe, on aurait noeud 1 → noeud 2 → noeud 1 ...

10. Notre système d'évènements

À chaque modification du modèle, ses vues doivent être notifiées qu'un changement a eu lieu afin de réagir en conséquence.
Notre choix s'est tourné vers une fonction générique qui utilise toute la puissance de Javascript.

Figure II.10.1 : Fonction permettant de notifier toutes les vues du modèle

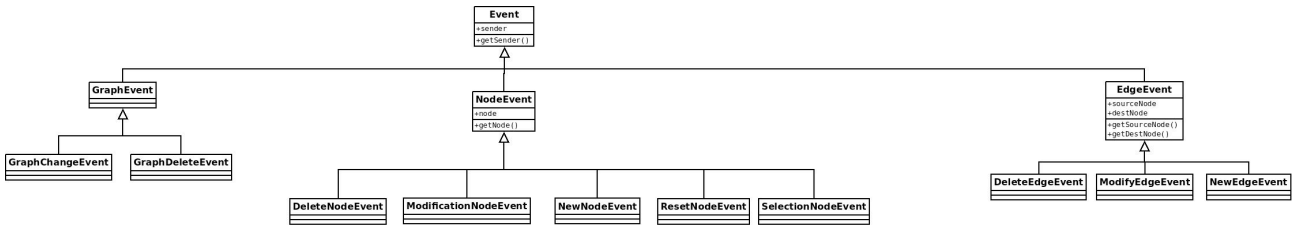
```
/** Fonction permettant de notifier l'ensemble des vues du modèle graph **/

AbstractModel.prototype.fireEvent=function(functionName,ev){ // Permet d'appeler le rafraichissement sur chacune des vues
  if(this.connectView){
    var i,m;
    for(i=0;i<this.getNumberOfView();i++){
      m=this.views[i][functionName];
      if(m!==undefined){
        this.views[i][functionName](ev);
      }
    }
  }
}
```

Ici on passe en argument le nom de la fonction à appeler sur les vues. Puis si elle existe on appelle la fonction demandée de la manière suivante :
« view[«nom de la fonction»]()» ce qui est équivalent à : «view.nom_de_la_fonction()».
De plus la fonction ne sera appelée que sur les vues la possédant, autrement dit même si pour une modification du modèle, certaines vues n'ont pas besoin de se rafraîchir il n'y a pas besoin de penser quelles vues il faut notifier, la fonction notifiera uniquement les vues possédant la fonction appelée.

On remarque également que cette fonction prend un deuxième argument : L'évènement. Les évènements sont des objets que nous avons crée afin d'apporter à la vue des informations supplémentaires, comme le modèle qui a demandé le rafraîchissement ou encore le nouveau noeud qui vient de s'ajouter au graph etc...

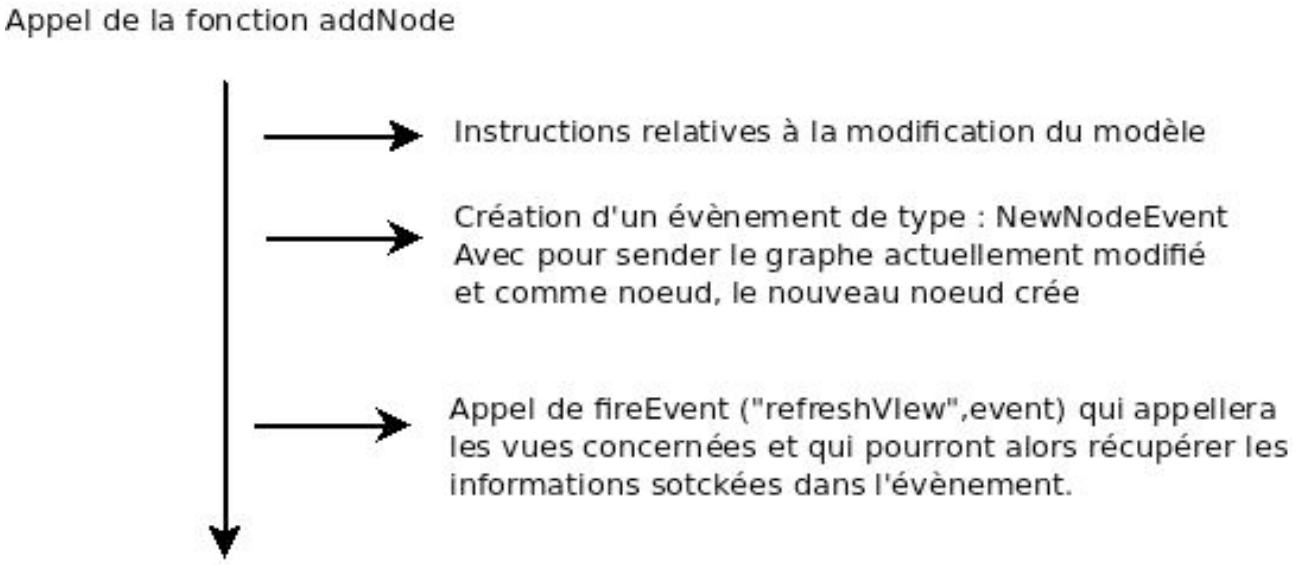
Figure II.10.2: Diagramme UML représentant notre implémentations des événements



Chaque évènement est associé à la notification d'un changement du modèle comme l'ajout ou la suppression d'un noeud.
Le fait de stocker des informations relatives à l'évènement permet ainsi d'avoir toutes les données nécessaires dans la fonction qui assurera le rafraîchissement de la vue.

En exemple voici les différentes étapes de l'ajout d'un noeud afin de mieux illustrer nos propos.

Figure II.10.3 : Schéma expliquant les différentes étapes de propagation d'un évènement (support : la fonction addNode())



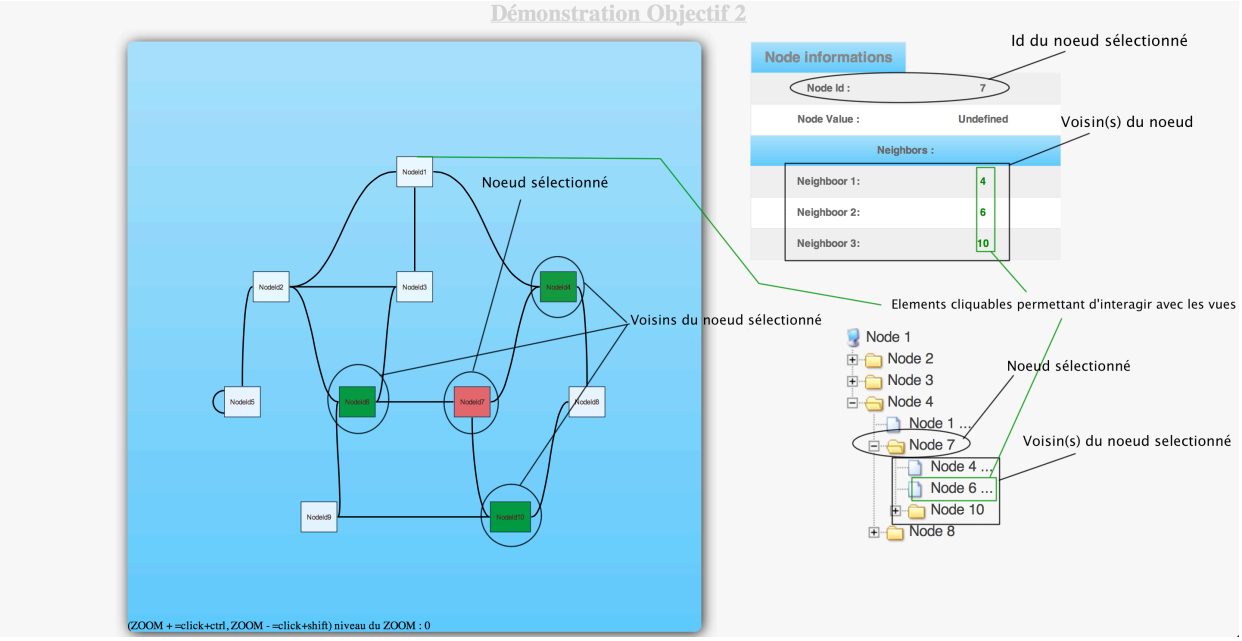
Remarque:
Il est possible de "déconnecter" les vues d'un modèle afin qu'elle ne soit plus notfier. Par exemple lors de la phase de création du graph il est inutile que les vues se rafraîchissent à chaque création de noeud ou d'arcs. En effet on ne cherche qu'à afficher le graphe final. En effet si l'on ne déconnecte pas les vues et que l'on crée un graphe de 2000 noeuds, 2000 rafraîchissement sur les vues concernées seront effectuées alors que cela ne sera même pas visible pour l'utilisateur.

Pour cela deux méthodes sont disponibles dans les objets de type AbstractModel :

- connectViews.
- disconnectViews.

11. Interactions entre les vues

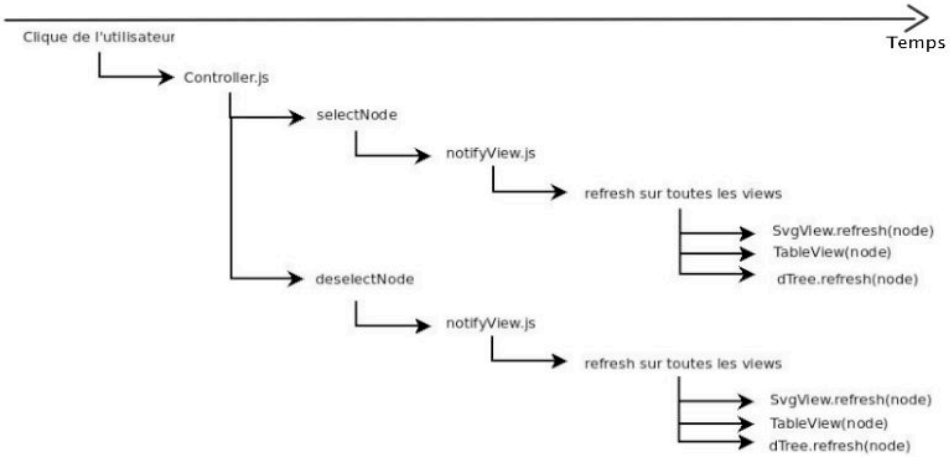
Figure II.10.1 : Fonctionnement de l'interaction entre les vues côté utilisateur:



L'utilisateur a donc la possibilité d'interagir avec chaque vue qui sont toutes «connectées» entres elles.
Lors d'une sélection d'un noeud dans la vue graphique, la vue texte se met à jour (changement de l'id, des voisins etc...) ainsi que la vue arborescente qui s'ouvre jusqu'au noeud sélectionné.
Chaque vue interagit donc avec les deux autres.

Ceci est possible grâce à l'implémentation du patron MVC que nous avons décris plus haut.
Le schéma ci-dessous explique comment nos vues interagissent du point de vue de notre implémentation :

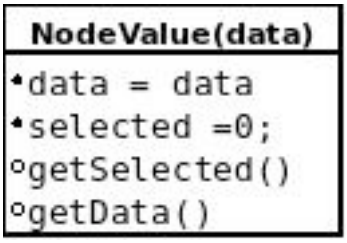
Figure II.11.2: Voici ce que engendre un clique de l'utilisateur



Lors d'un clique de l'utilisateur, les écouteurs contenus dans le contrôleur appelle les méthodes selectNode(id) / deselectNode(id) du modèle.
L'id est ici récupéré grâce à jQuery en accédant au DOM, on récupère l'attribut «NodeId» du noeud sélectionné.
Les méthodes du modèle permettant de sélectionner ou de désélectionner un noeud modifie l'état de sélection de ce dernier. En effet nous avons légèrement modifié le modèle en changeant l'attribut node.value = integer en node.value = new NodeValue (integer).

Ci-dessous la représentation de cette nouvel objet :

Figure II.11.3: Objet NodeValue

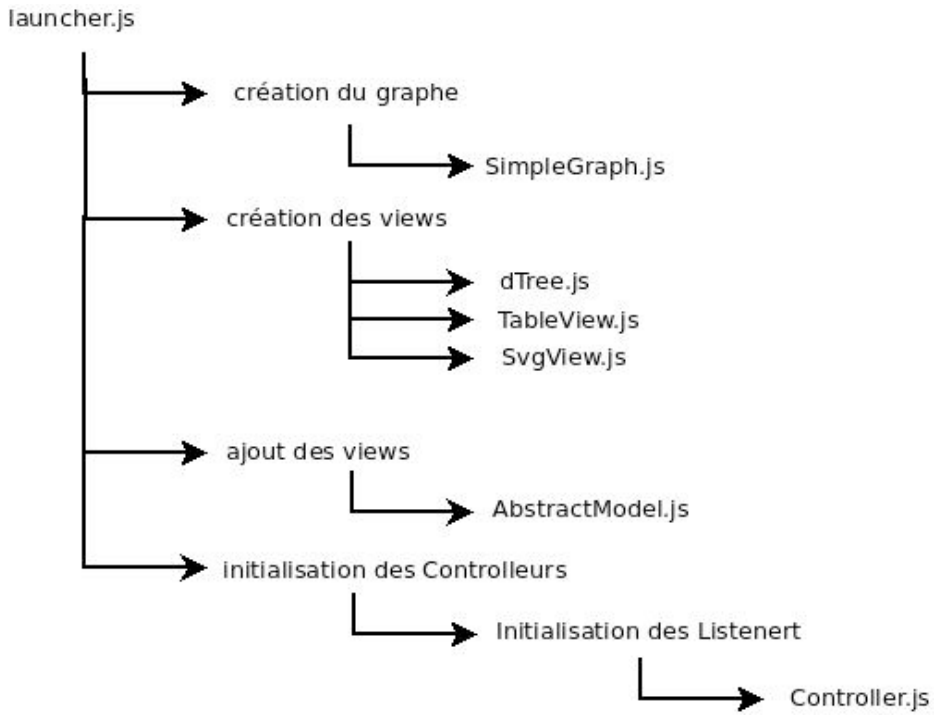


Ainsi on ne modifie pas directement le modèle et il est alors facile d'aller indiquer le fait qu'un noeud soit sélectionné en tant que noeud courant ou noeud voisin du noeud couramment sélectionné.

Les méthodes selectNode/deselectNode appelle alors la méthode notifyView() qui se charge d'informer toutes les vues qu'un changement a été effectué et appelle alors la méthode refresh(node) de chaque vue en leur indiquant quel noeud a été modifié.
Les méthodes refresh de chaque vue opèrent alors les traitements nécessaires à leur mise à jour.

12.Chargement d'un graphe

Figure II.12.1: Schémas représentant le chargement d'un graphe



Notre launcher.js est le fichier permettant d'initialiser les vues de notre graph.
A l'intérieur on y crée notre graphe (noeud, relations entre eux .. etc) ainsi que la création des différentes vues (vue tableau,vue arborescente, vue graphique) qui sont alors ajoutés au modèle(graphe).
Le contrôleur est lui aussi créé afin de pouvoir appeler sa méthode initializeListenners qui permet d'ajouter tous les écouteurs nécessaires à l'interaction entre les vues.

13.Conclusion

Rappel des exigences:

Exigences	Description	Couverture
5	Votre plugin doit être implémenté selon le patron de conception MVC (Modèle Vue Contrôleur)	85 %

Exigences	Description	Couverture
10	Multi-vues. Votre interface graphique doit comporter deux vues sur le graphe : – Une vue « texte » avec une ligne par nœud. Chaque ligne doit indiquer l'id du nœud, sa valeur, et l'id de ses voisins. – Une vue graphique illustrant les nœuds et les arcs du graphe. La représentation des nœuds et des arcs est laissée libre mais l'utilisateur doit être capable de retrouver l'ensemble des informations sur le graphe.	100 %
11	La vue graphique devra utiliser la technologie SVG.	100 %
12	TreeLayout. La vue graphique posséder un gestionnaire de placement (LayoutManager) de type « TreeLayout », permettant d'organiser automatiquement des nœuds sous la forme d'un arbre (tel qu'illustré dans la figure 3).	95 %
13	Interactions avec l'utilisateur. La vue graphique devra au minimum permettre de : – déplacer des nœuds – translater la représentation (outil « main ») Et idéalement de zoomer/dézoomer.	100 %

Nous pensons avoir réussi globalement l’ensemble des objectifs. Néanmoins quelques limites sont apparus notamment lors des tests vu précédemment ou encore au niveau du MVC où nous sommes venu ajouter des méthodes à notre modèle défini

dans l'objectif 1.
Nous essayerons de corriger ces défauts pour le rendu du prochain objectif.
Au niveau de l'interface nous pensons avoir crée quelque chose d'intuitif pour l'utilisateur qui peut interagir facilement avec chaque vue. Ne perdant pas l'objectif final du projet qui est rappelons le d'aider les débutants en programmation de comprendre les différents liens dans les fonctions.

Objectif 3: Intégration

Pour cette troisième et dernière partie du projet, le but est d'intégrer notre travail à AlgoView afin d'atteindre le but principal de ce projet, qui est rappelons le, l'affichage du "CallGraph".
Dans cet objectif trois exigences sont à respecter:

- Fonctionnement en mode «ProgramTree»
- Fonctionnement en mode «CallGraph»
- Une interface permettant de passer facilement d'un mode à l'autre

14. Fonctionnement du compilateur Algoview

Afin de mener à bien cet objectif, les fichiers du compilateur d'AlgoView ont été mis à notre disposition. La compilation d'un programme est relativement simple :
On appelle la fonction ***buildProgramTree(****fileName****)*** avec pour argument le nom de notre fichier écrit en simple language (il est à noter que les fichiers de ce type on une extension .sl).
La fonction effectue alors une requête AJAX (acronyme d'Asynchronous JavaScript and XML) afin d'aller rechercher le fichier précisé en argument. En cas d'erreur la fonction *loadErrorHandler* est appelée, cette dernière vient alors afficher un message d'erreur à l'écran.
En cas de succès de la requête c'est la fonction ***compileProgram***(*programText*) qui est appelée. Cette dernière exécute la fonction *compile* de l'objet ProgramCompiler, fonction qui a pour but de détecter toutes erreurs de syntaxes ou du langage comme le fait n'importe quel compilateur dans un autre langage.
Si la compilation s'est correctement déroulée, on peut alors accéder à "l'arbre du programme". Stockée dans l'instance courante de l'objet ProgramCompiler il est composée de toutes les informations du programme (Appel de fonction, déclaration de variable, assignation d'une variable …). Au total plus de cinquante types de noeuds existent. C'est l'arbre le plus complet qui puisse être puisqu'il contient tout ce qui se passe dans le programme. Néanmoins même si l'objectif est d'avoir un graph détaillé du programme certaines informations ne sont pas nécessaires à la compréhension du programme pour des étudiants de première année.
C'est là, l'objectif du mode ProgramTree que nous allons détailler dans la prochaine partie.

15. Le mode ProgramTree

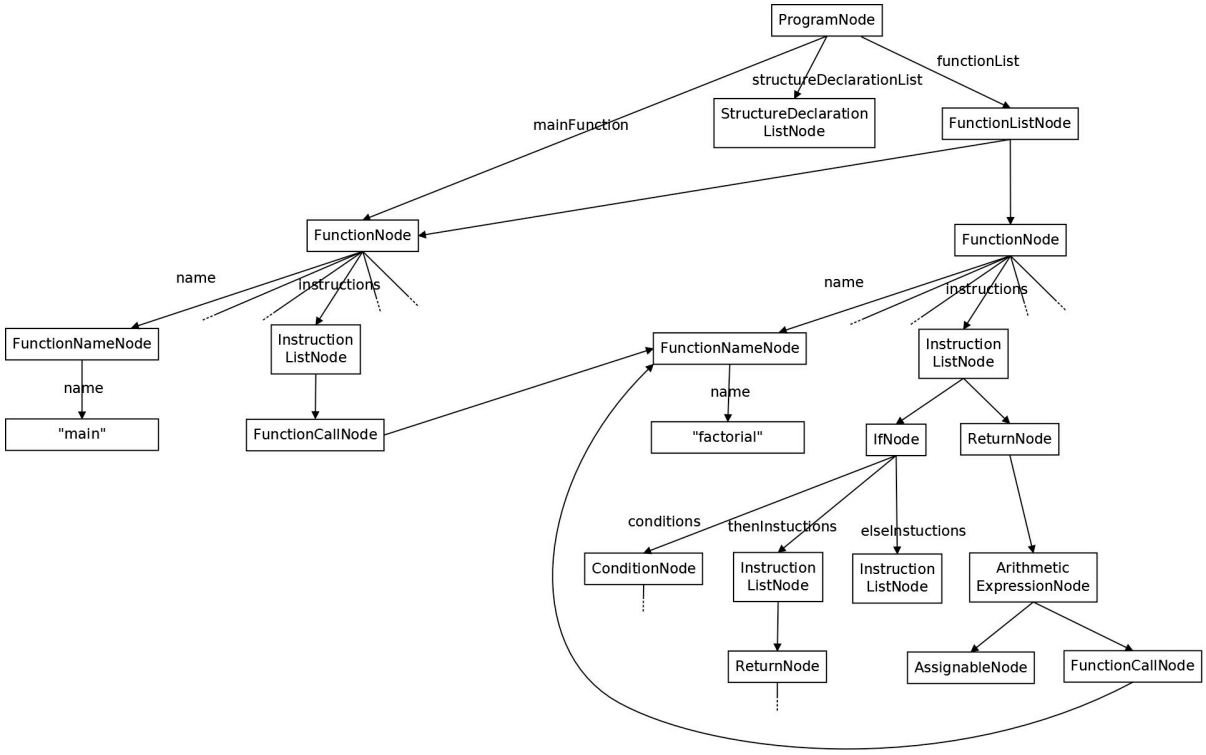
- **Présentation**

  Il consiste d'une part à passer de l'arbre récupérer par à la suite de la compilation à un graph comme nous l'avons vu dans les précédents objectif afin qu'il puisse fonctionner dans notre application et d'autre part à ne garder comme informations que les types des noeuds rencontrés. Ce mode est donc une version "allégé" de l'arbre renvoyé par le compilateur néanmoins le fait qu'il soit allégé ne veut pas dire moins complet, en effet il faudra veiller à garder pour ce mode l'intégralité des noeuds présents dans l'arbre.
  Prenons un exemple afin de mieux comprendre ce que l'on attend comme résultat, soit le programme ci-dessous :

Figure III.15.1: Implémentation de factoriel sur Algoview en récursive:

```
1  FUNCTION factorial(i: INTEGER) : INTEGER
2  BEGIN
3      IF (i<= 1) THEN
4          RETURN 1
5      END_IF
6      RETURN i*factorial(i-1)
7  END
8
9  PROCEDURE main()
10 BEGIN
11     factorial(3)
12 END
```

Figure III.15.2: Le résultat voulu de notre application en mode ProgramTree



- **Implémentation**

Afin de parvenir à ce résultat nous avons réalisé la fonction ***ConvertTreeInGraph***(*root*) qui permet de convertir un arbre en un graph. Ici comme nous ne voulons garder que le type des noeuds nous stockerons dans le nodeValue.name de chaque noeud leur type.
Le compilateur renvoie le premier noeud de l'arbre, néanmoins nous avons accès à ses voisins il est facile d'effectuer un parcours en largeur de l'arbre afin de récupérer tous ses noeuds. Pour chaque nouveau noeud visité dans l'arbre nous lui créons son équivalent dans un graphe en y stockant son type.
Ci-dessous les schémas expliquant le déroulement du parcours et la création du graphe associé à l'arbre.

Figure III.15.3 : Explication de la visite du premier noeud de l'arbre

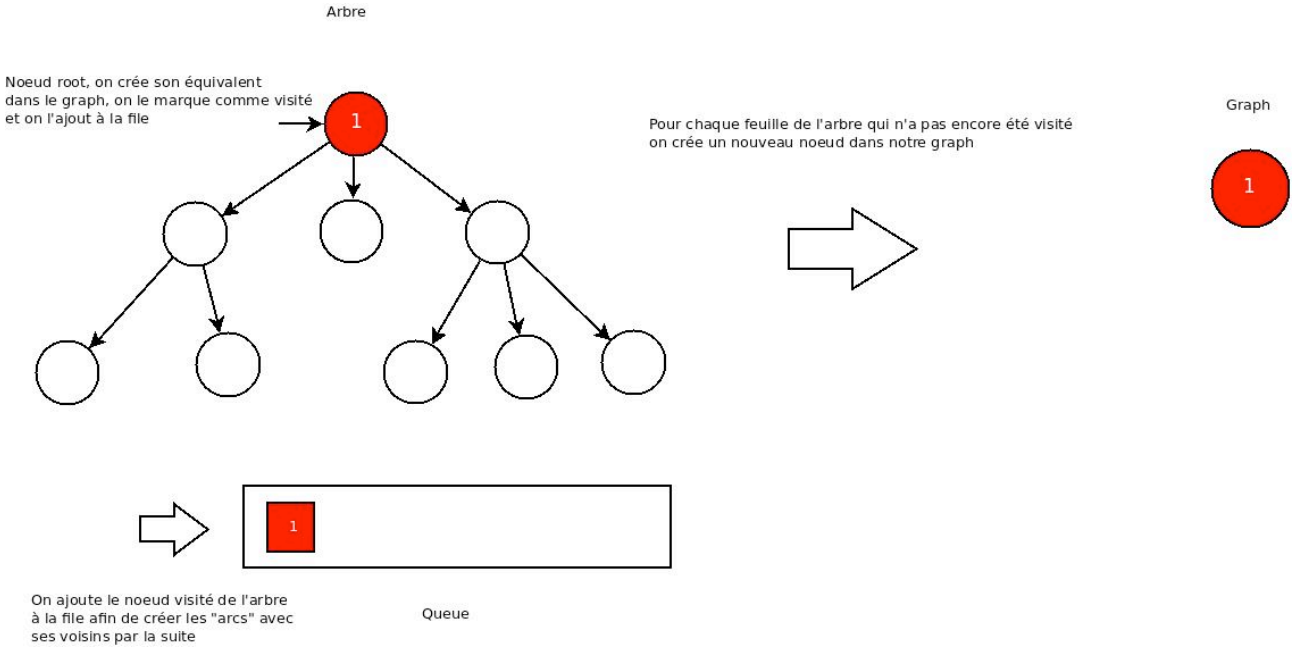
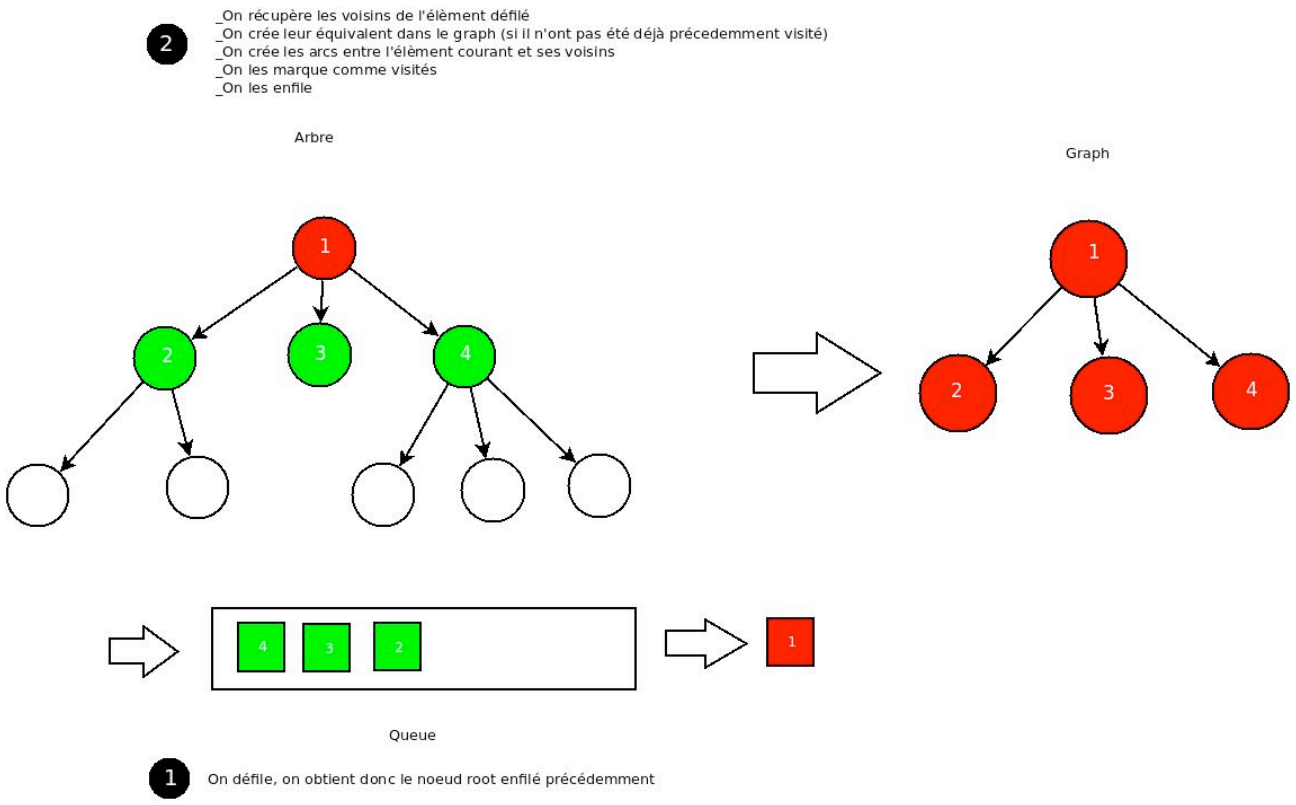


Figure III.15.4 : Explication de la suite du parcours(défilement d'un noeud, création de ses voisins et des arcs avec ces derniers)



On continue l'opération illustré en *Figure III.14.4* jusqu'à ce que la file soit vide, les noeuds auront alors été tous visité. On récupère alors le graphe crée.
Pour l'afficher il ne restera alors plus qu'à effectuer les mêmes opérations que celles expliquées dans l'objectif 2 (création des vues … etc).
Passons maintenant au deuxième mode que nous avons du implémenter et qui est le but principal du projet : le mode «CallGraph».

16. Le mode CallGraph

- **Présentation**

Le but de ce mode est de créer un graphe composé uniquement des fonctions appelées dans le programme afin de mieux comprendre le déroulement celui-ci. Il s'agit d'une version encore plus "allégée" que le mode ProgramTree.
Les noeuds afficheront le nom des fonctions exécutées, les fonctions déclarées dans le programme mais n'étant pas appelée n'apparaîtront pas dans notre vue graphique.
L'utilisateur aura alors une vue d'ensemble sur son programme ainsi que les liens entre les différentes fonctions.

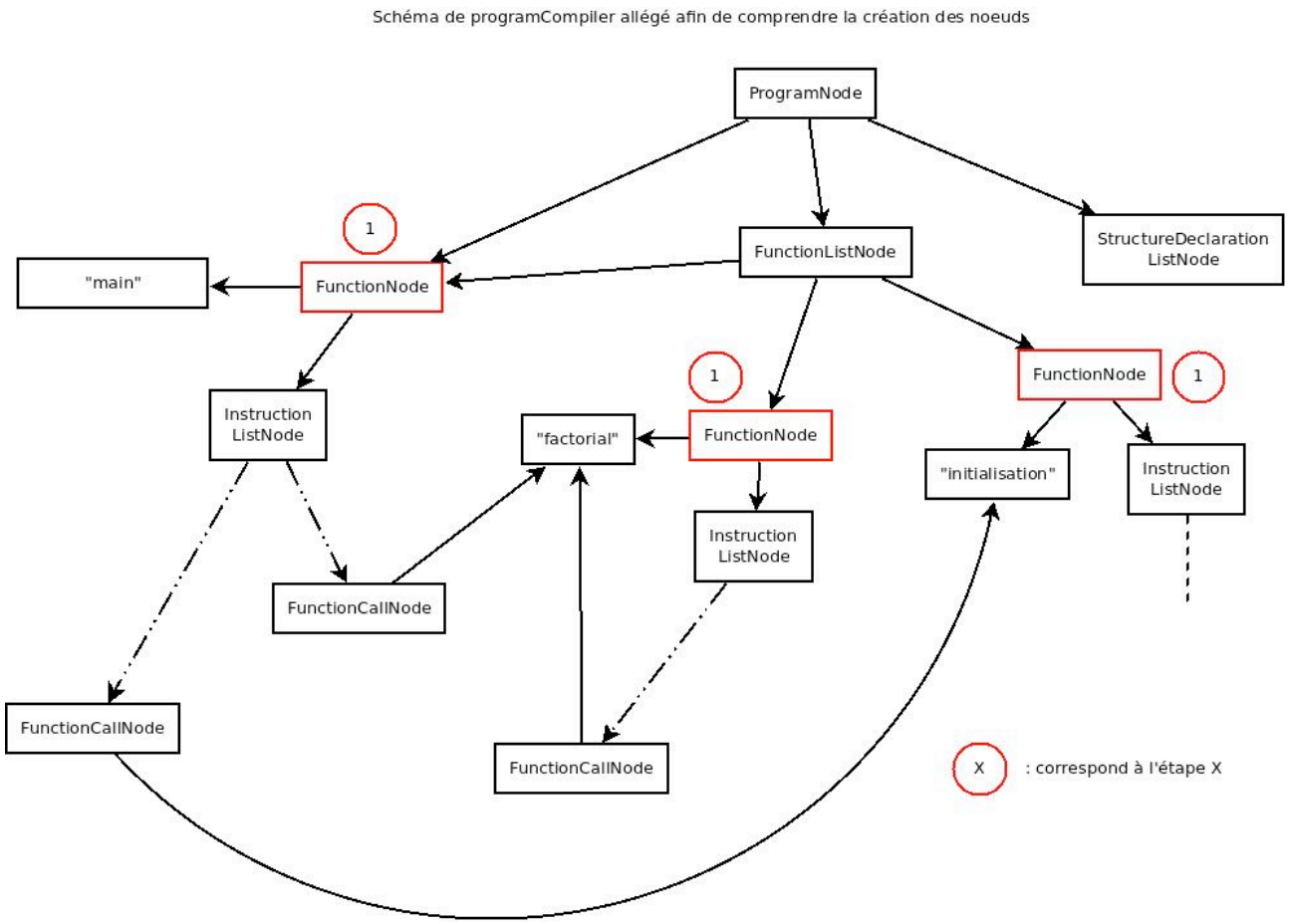
- **Implémentation**

La création du graphe « allégé » se fait en 3étapes dont 2 étapes qui vont se répéter souvent.

★ La création des noeuds

Lorsqu'on récupère l'arbre renvoyé par le compilateur, nous avons toutes les fonctions et procédures implémentées dans le programme stockées dans un noeud : FunctionListNode. Ainsi pour chacune des fonctions ou procédures, nous pouvons créer un nœud et l'ajouter à notre graphe « allégé ». Le nœud est crée avec le nom de la fonction dans son champ value ainsi qu'avec un id unique incrémenté automatiquement dans la fonction.

Figure III.16.1: Schéma représentant le lien entre le noeud de type FunctionListNode et les noeuds de type FunctionNode dans l'arbre renvoyé par le compilateur

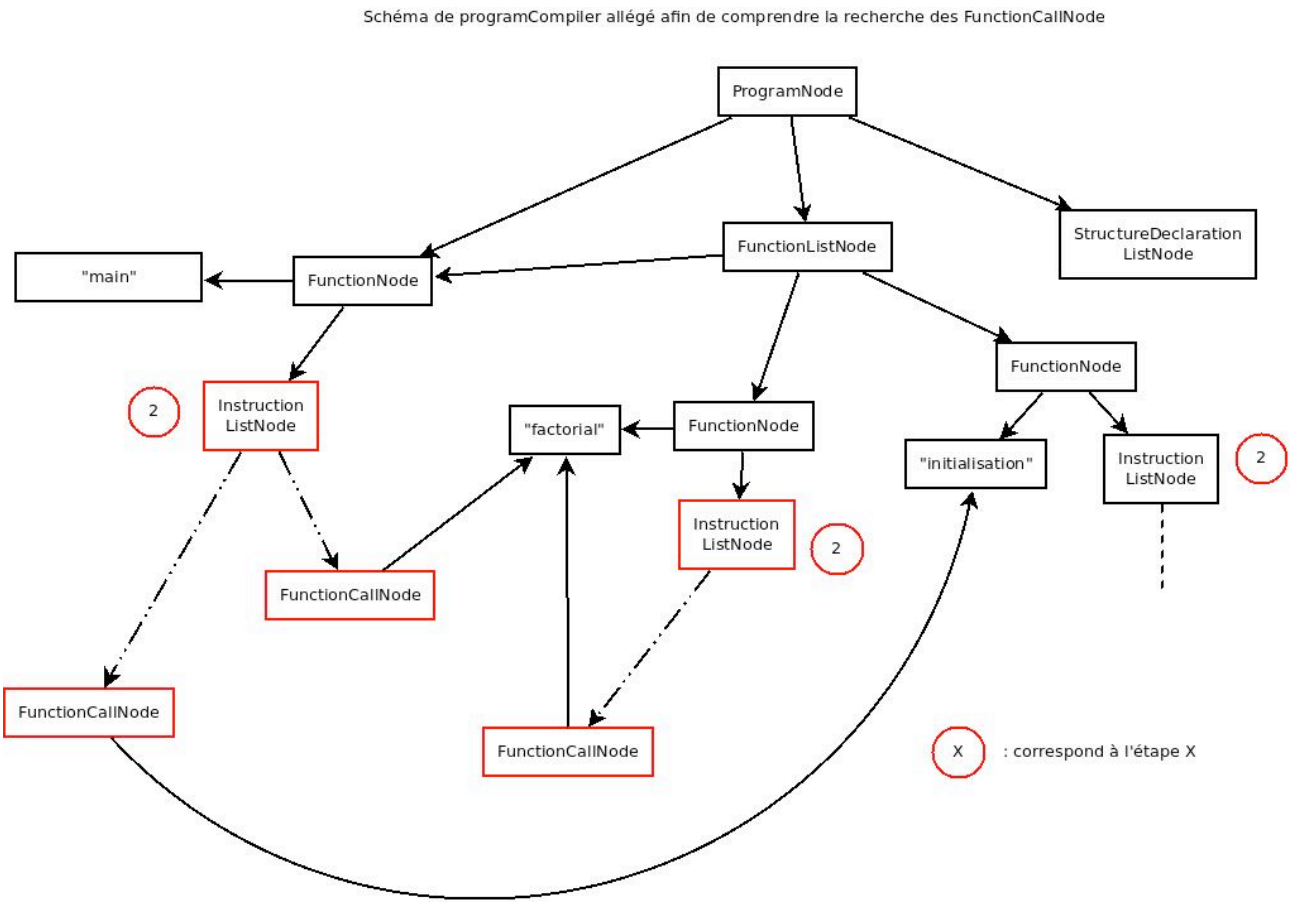


★ Recherche de arcs

Pour chacun des nœuds, nous allons regarder si il fait appel à d'autres fonctions.Pour cela, dans programTree, il existe un nœud, FunctionCallNode, qui permet de savoir qu'il y à un appel de fonction. Le but est de rechercher toutes les FunctionCallNode ayant pour ancêtre le noeud en question. En effet si dans le code d'une fonction une autre fonction est appelé il y aura alors dans l'abre un nouveau noeud à aller rechercher.

Pour se faire, dans l'abre, chacun des nœuds FunctionNode (représentant les fonctions), possède un nœud, InstructionListNode, comportant tout le code dans des nœuds sous-jacent. Le but est de parcourir ces nœuds afin de trouver les FunctionCallNode de la fonction.
Une fois qu'on les a tous parcouru, on s'arrête là en effet on sait que la fonction appelée possède déjà un nœud dans le graphe « allégé » et qu'il a été ou sera traité plus tard dans le code.

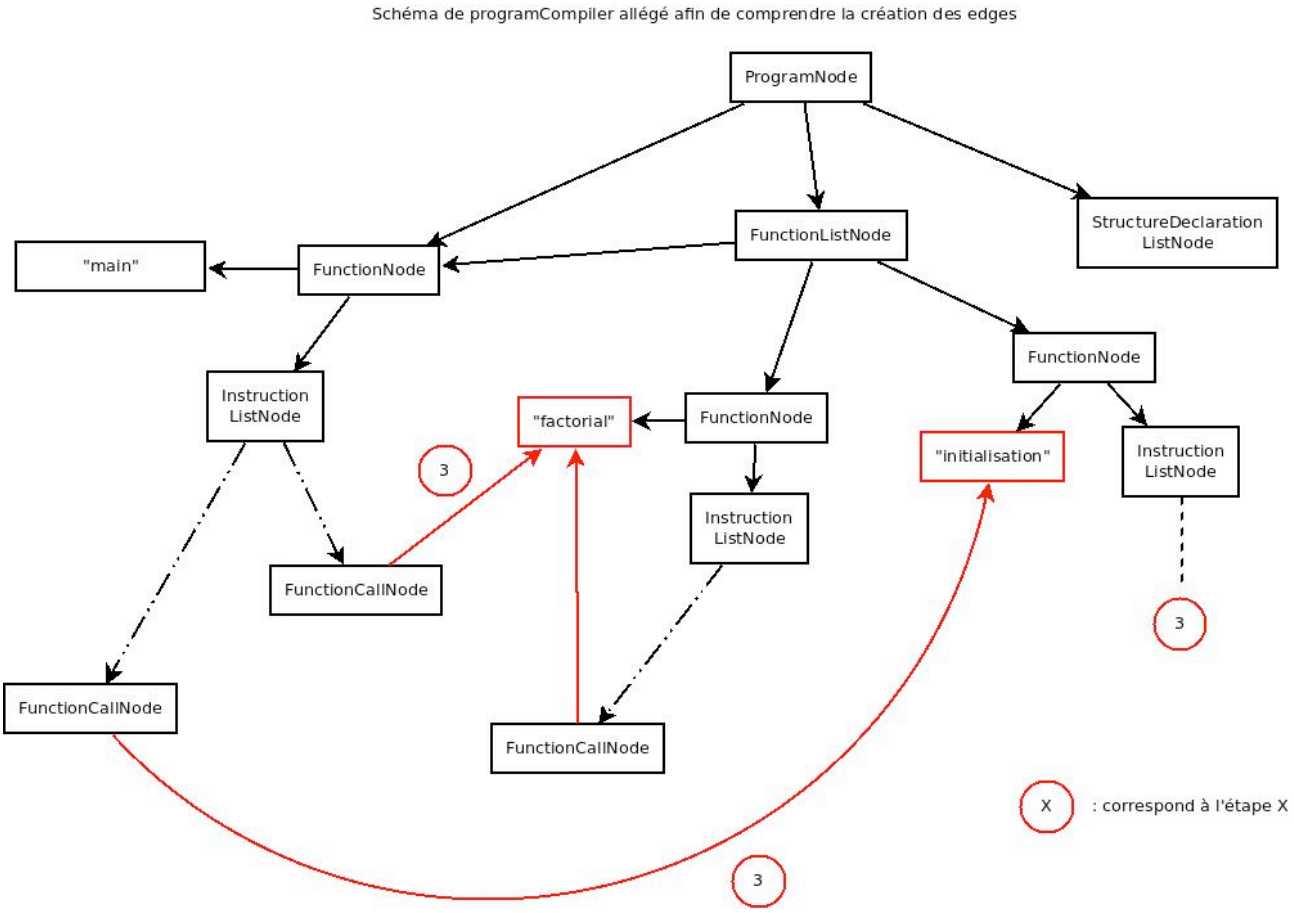
Figure III.16.2: Schéma expliquant les dépendances entre les noeuds de type «FunctionNode» et ceux du type «FunctionCallNode»



★ Création des arcs

Une fois que l'on a récupéré tout les appels de fonction, on crée l'arc permettant de symboliser que la fonction fait appel à une autre (voir plusieurs). Il faut veiller à préciser que le graph est direct en effet l'appel d'une fonction à l'intérieur d'une autre ne se fait que dans un sens. Pour créer l'arc, nous avons implémenter une fonction qui prend en paramètre l'id du nœud (celui que nous avons donné à chacun des nœuds dans le graphe « allégé » de façon unique) afin de faciliter la création des arcs.
Au préalable nous avons stocké dans un tableau tout les ids des noeuds ainsi que la fonction correspondante à l'id dans le graphe. Il est alors facile de retrouver le noeud du graphe qui correspond à tel ou tel fonction.

Figure III.16.3: Schéma représentant comment récupérer la fonction «parente» d'un noeud de type «FunctionCallNode»



★ Remarque

Lors de la création du graphe « allégé », nous avons pu lui faire ajouter des nœuds qui n'auront aucune correspondance avec d'autres dans le graphe, c'est à dire des fonctions implémentées mais non utilisées dans le code de l'utilisateur. Ces derniers seront alors traités plus tard lors de la création de la vue graphique : SvgView, qui n'affichera pas les nœuds sans correspondance avec d'autres nœuds aussi bien comme parent ou fils d'un autre nœud.

★ Exemple

Afin de mieux comprendre le fonctionnement, nous allons plus détailler les schémas qui ont servi de support au préalable
Voici le code qui permet la construction du graphe en exemple :

```
FUNCTION factorial ( i:INTEGER) : INTEGER
BEGIN
        IF( i=1) THEN
                RETURN 1
        END_IF
        RETURN i*factorial(i-1)
END
```

```
PROCEDURE initialisation ()
BEGIN
        PRINT(« calcul de la factorielle en cours »)
END

PROCDEDURE main()
BEGIN
        initialisation()
        factorial(3)
END
```

Lors de la première étape du code, nous créons les nœuds dans le graphe «allégé».

Figure III.16.5

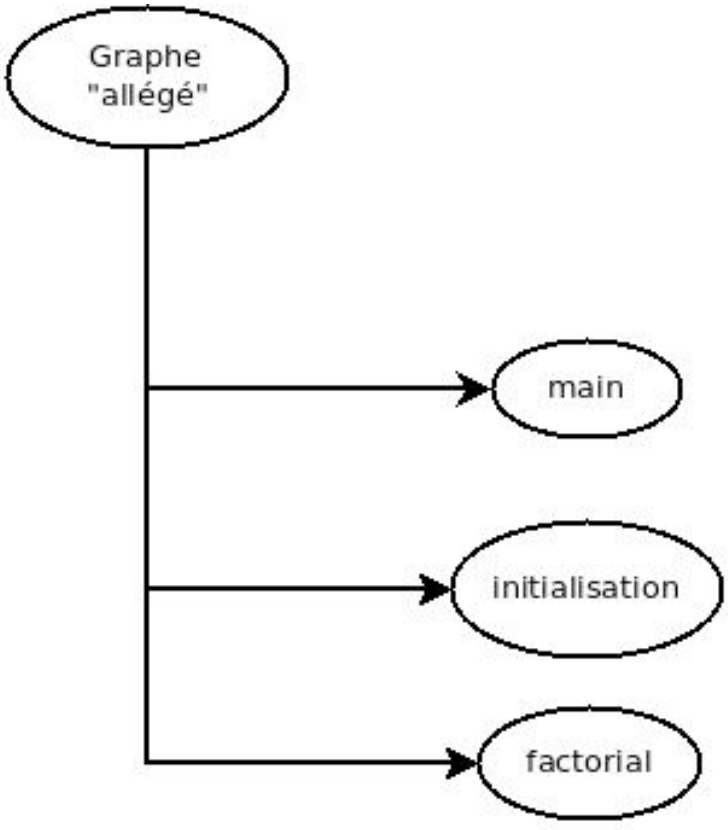
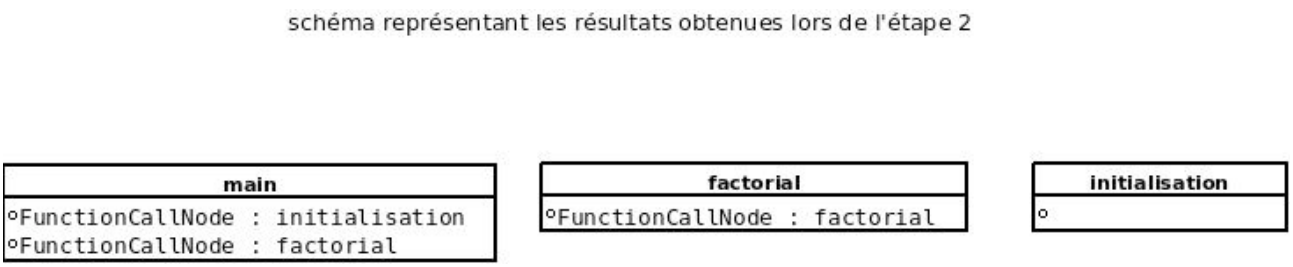


schéma représentant la vue du type graphe du graphe "allégé"

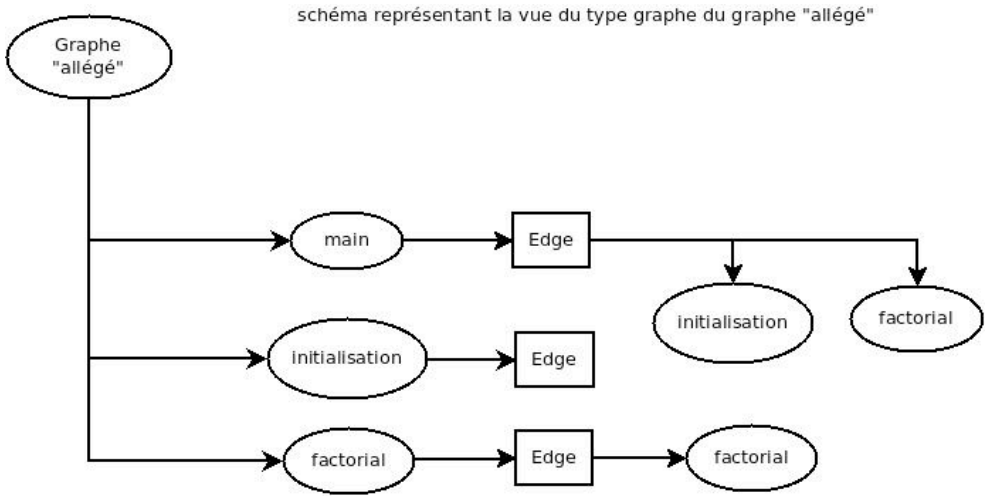
Ensuite nous cherchons pour toutes les fonctions leurs appels de fonctions.

Figure III.16.6:



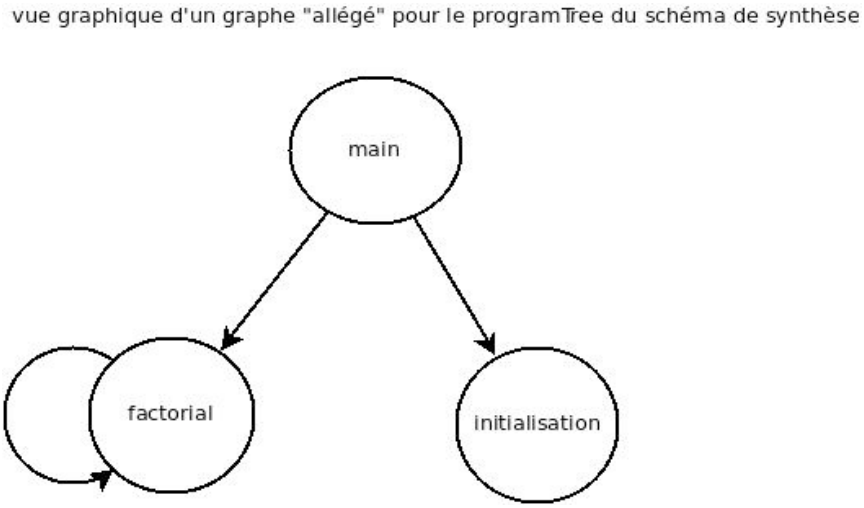
Enfin nous créons les arcs permettant des les relier :

Figure III.16.7:



Du côté de la vue graphique ce programme doit nous donner en mode CallGraph :

Figure III.16.8:



17. Passage d'un graphe à l'autre

Dans les spécifications il nous était également demandé de pouvoir passer facilement d'un mode à l'autre, l'interface et le design étant laisser libre.
Le respect du modèle MVC prend ici tout son sens, en effet ici le fait de changer de mode revient à changer le modèle au niveau du controleur qui demande alors au modèle de rafraîchir ses vues . La propagation de l'information jusqu'à ces dernières est alors assurer par notre implémentation du MVC.
Ci-dessous, deux schémas qui aideront à la compréhension du déroulement en interne lors du passage d'un mode à l'autre.

Figure II.17.1: Schémas expliquant le déroulement en internet lors du passage d'un mode à l'autre.

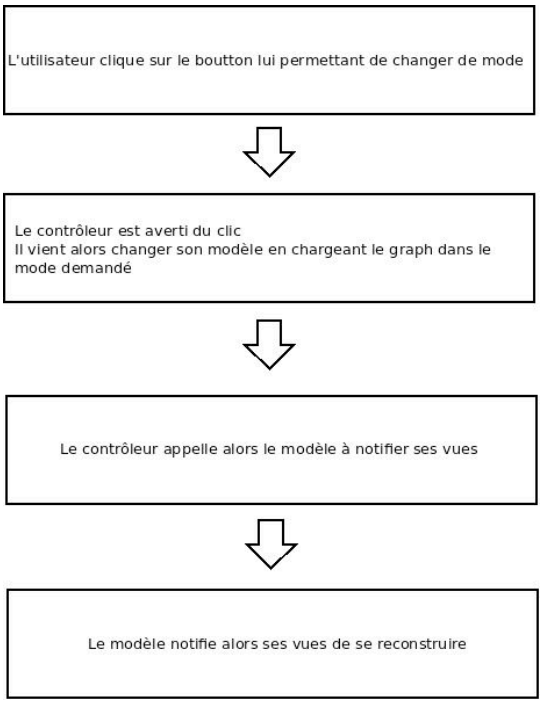
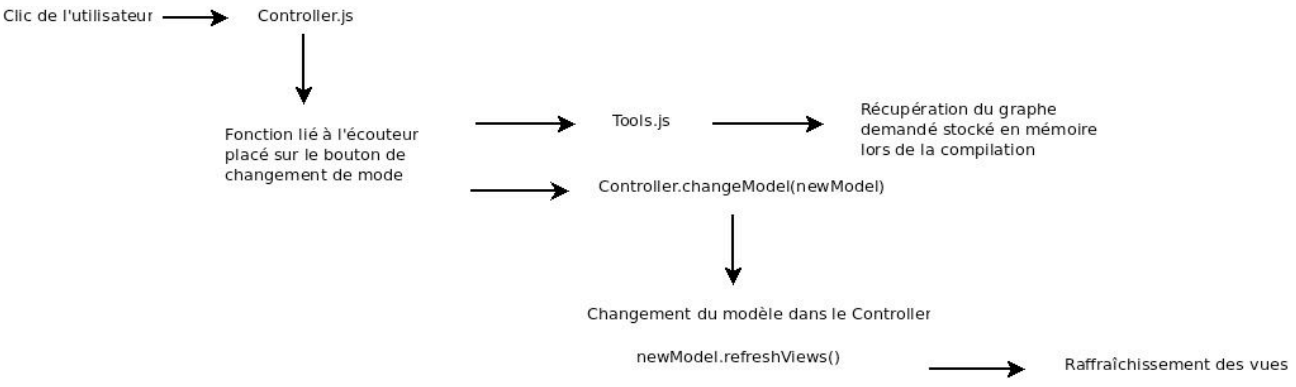


Figure II.17.2



18. Conclusion

Nous sommes globalement satisfait du résultat obtenu lors de cet objectif, nous avons testé plusieurs programmes les résultats en mode callGraph sont ceux attendus. Au niveau du ProgramTree il devient vite difficile de vérifier si le graphe est bon ou non vu le nombre de noeud présent.
On comprend ici tout l'importance du MVC lors du passage d'un graphe à un autre qui permet de rafrâichir quasi-automatiquement toutes nos vues.
Quelques limites néanmoins apparaissent directement liés à celles de notre TreeLayout du précédent objectif notamment lors de l'affichage du ProgramTree de programme constitué de plus d'une fonction où le graphe devient rapidement illisible du au nombre important de noeuds.

Conclusion Générale

19. Couverture des exigences

Exigences	Couverture
Support des navigateurs(Firefox/chrome)	100 %
Structuration des fichiers	100 %
Contenu de la page principale	100 %
Répartition des scripts	100 %
Patron MVC	100 %
Variables globales	80 %
Programmation Orientée Objet	85 %
Respect du type abstrait	100 %
Implémentation concrète	100 %
Multi-vues	100 %
SVG	100 %
TreeLayout	90 %
Interactions avec l'utilisateur	100 %
Affichage de l'arbre modélisant un programme	100 %
Fonctionnement en mode «ProgramTree»	100 %
Fonctionnement en mode «CallGraph»	100 %

20. Difficultés rencontrées

Tout au long du projet plusieurs problèmes se sont posés. Tout d'abord dès le premier objectif le choix de l'implémentation de notre type Graphe était une décision importante. En effet le modèle étant la représentation des données en mémoire son choix de conception est la base du projet et doit être bien pensé afin d'éviter d'engendrer d'autres problèmes plus tard dans le projet. Ayant vu lors des séances de TD plusieurs façons d'implémenter les graphes il a fallu faire un choix tout en réfléchissant aux besoins que nous aurions dans l'avenir du projet. Notre objectif étant de construire un graphe d'appel des fonctions, ce dernier n'aura pas besoin d'être sans cesse modifié, en revanche nous aurons régulièrement besoin d'aller chercher des informations relatives aux noeuds. C'est donc là toute la difficulté, prendre des décisions au niveau de l'implémentation en pensant déjà aux futurs objectifs.

Une autre point à aborder est celui de la technologie SVG et de la bibliothèque Raphael, en effet sa prise en main fut longue et fastidieuse n'ayant jamais abordé le SVG il a fallu d'abord comprendre son fonctionnement avant de pouvoir réellement commencer à utiliser Raphael. De plus des problèmes qui relevaient plus des mathématiques se sont posés comme pour le zoom ou la translation de notre vue graphique.
Enfin un point également important est celui de l'organisation dans le groupe, savoir définir le travail à faire pour chaque objectif ainsi que sa répartition pour chaque membre. Une capacité qui est primordiale dans notre futur vie d'ingénieur.

21. Enrichissement personnel

- **Alexis Dufour**

  Cette année nous avons découvert un nouveau langage, le JavaScript, celui ci-étant un langage majeur pour le projet.
Ce dernier m'a permis de découvrir toute la puissance de ce langage, même si nous n'avons, je pense, pas exploité la totalité de ses capacités, il fut par moment difficile de résoudre certains «bugs» de code par un manque de connaissances dans ce langage, ce qui m'a appris à aller plus loin dans mes recherches et demander de l'aide au professeur ou aux autres membres du groupe. Au cours de ce projet j'ai pu découvrir concrètement le MVC ce qui m'a permis de mieux comprendre ce pattern.

  D'un point de vue organisation, je pense que nous nous sommes très bien organisé, nous avons su bien départager la masse de travail entre les membres du groupe et nous avons toujours réussi à finir dans les temps.

- **Nicolas Olive**

  Au cours du projet, j'ai pu tester mes connaissances mais aussi découvrir de nouveaux langages et de concepts. Ce projet m'a apporté beaucoup de connaissance, j'ai pu améliorer mes compétences en matière de JavaScript et revoir les principes d'un MVC. J'ai été de nombreuse fois confronté à la limite de mes connaissances ce qui a eu pour effet, d'aller chercher de nouvelles choses et de principes en matière de code tout en respectant les exigences demandés au cours du projet.
De plus, ce projet étant réalisé en groupe, ma permis découvrir ce que c'était le travail d'un projet en groupe et le mener à son terme. Chacun d'entre nos a pu exprimer son idées en matière d'implémentation du code, et nous avons pris le meilleur de chacune de nos idées. Le fait d'avoir des rendus et des exigences au cours du projet, m'a permis de structurer le code mais aussi dans le concept, étape par étape, afin de réaliser le projet et aussi de respecter un rythme de travail et d'organisation.

- **Matthieu Delmaire**

  Au tout début de l'année je n'avais encore jamais utilisé le langage Javascript. C'est lors du premier semestre grâce aux cours de Monsieur Sablonnière que je découvris ce langage "particulier" qui ne ressemblait pas aux autres langage objets comme le C++ ou le PHP. De plus je pensais que le javascript était essentiellement un langage destiné à rendre des pages webs dynamiques. Or j'ai découvert qu'il était beaucoup plus puissant que ça. Avec ce projet j'ai découvert que le Javascript était un langage à part entière qui de part ses particularités le rendait parfois plus difficile à manipuler mais qu'il pouvait également se révélait très puissant permettant de réaliser des choses impossibles dans les autres langages vu jusqu'à présent dans mon cursus à l'ISEN.

Le deuxième point important a été la réelle compréhension du pattern MVC au cours de ce projet. En effet l'année dernière lors du projet WEB de première année, ce concept avait été très peu abordé et ses avantages et son fonctionnement me paraissaient très flous. Avec ce projet CallGraph j'ai pu donc comprendre la nécessité et les atouts du MVC.
Un point qui me parait également important est celui de la découverte de la librairie Raphael ainsi qu'une technologie que nous n'avions encore jamais abordé : le SVG. Le fait de comprendre le fonctionnement d'une bibliothèque, de ses fonctions et surtout comment elle pourra s'adapter dans son application est une chose qui je pense est essentiel dans une époque ou l'open source se développe de plus en plus et où il et parfois très utile et même indispensable d'économiser un temps précieux d'utiliser des bibliothèques mise à disposition gratuitement sur internet.
Enfin ce projet a été également une bonne expérience , au niveau de l'organisation dans un groupe ainsi que savoir se fixer des objectifs afin de respecter les dates de rendu.