|  |
| --- |
| **Rapport PSTL** |
| [Tapez le sous-titre du document] |
|  |
| [Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document. Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document.] |
|  |
| **Morvan Lassauzayet Victor Nea** |
| **30/06/2016** |
|  |

LES BOUTS DE PHRASES EN ROUGES SONT DES NOTES QUI DEVRONT ETRE SUPPRIMEES.

(PARTIES II ET III à refaire, il faudrait fusionner les deux parties)

(Il faudrait également renommer certaines sous parties)

Sommaire

[Introduction 3](#_Toc449627786)

[I - Présentation générale du projet 4](#_Toc449627787)

[1. Présentation DOT + GraphViz 4](#_Toc449627788)

[2. Veille concurrentielle 5](#_Toc449627789)

[3. Vue d'ensemble de l'application espérée 6](#_Toc449627790)

[4. Digramme de use case 7](#_Toc449627791)

[5. Outils choisis et pourquoi 8](#_Toc449627792)

[II - Architecture 9](#_Toc449627793)

[1. Architecture générale 9](#_Toc449627794)

[2. Architecture de chaque composant et description générale 9](#_Toc449627795)

[3. Liaison inter-composants: description plus fines car explication des méthodes choisies et de leur forme 9](#_Toc449627796)

[III - Interface et éléments graphiques 10](#_Toc449627797)

[1. Description des outils utilisés 10](#_Toc449627798)

[2. Gestion graphique des nœuds et arêtes 10](#_Toc449627799)

[3. Gestion vue textuelle 10](#_Toc449627800)

[IV - Maintenabilité et distribution 11](#_Toc449627801)

[1. Exemple d'extension 11](#_Toc449627802)

[2. PEP, licence, packaging 11](#_Toc449627803)

[Conclusion 12](#_Toc449627804)

[Tables des illustrations 13](#_Toc449627805)

[Annexes 14](#_Toc449627806)

Introduction

Dans le cadre de notre formation en 1ère année de Master spécialité STL, nous avons été amenés à développer une application nommée dotEd pour l'UE PSTL. (A compléter)

I - Présentation générale du projet

## 1. Présentation DOT + GraphViz

Le langage DOT est un langage de description de graphe dans un format texte (extension dot). Il est donc possible de définir des nœuds, des arrêtes, des sous-graphes, etc et de leur définir des attributs comme un label, une couleur, ..

Voici la grammaire du langage DOT (à mettre en annexe ou inutile peut-être ?) :

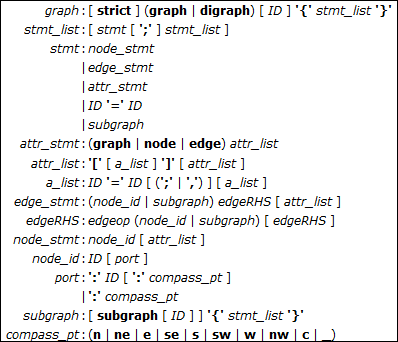


Figure 1: Grammaire du langage DOT

(source : http://www.graphviz.org/content/dot-language)

A l'aide du logiciel GraphViz (Graph Visualization Software), il est possible d'importer des fichiers DOT afin de visualiser le graphe sous forme graphique. Il est ainsi possible de sauvegarder ce rendu graphique sous forme d'image.

Voici un exemple d'un fichier DOT et de sa représentation graphique avec GraphViz :



Figure 2: Représentation graphique d'un fichier DOT

Utilité de notre logiciel :

GraphViz est un outil très puissant pour la génération graphique de graphe. Cependant il n'est pas possible de faire des actions (création, édition ou suppression d'un élément) directement sur la vue graphique comme c'est une image. Le seul moyen est donc de modifier le fichier DOT puis regénérer le graphe. Ce que nous souhaitons c'est donc de pourvoir modifier les éléments directement sur la vue graphique et sur la vue textuelle.

## 2. Veille concurrentielle

(retrouver lien Stack Overflow puis tester chaque logiciel).

## 3. Vue d'ensemble de l'application espérée

(screenshot ou maquette + description ?)

Voici une maquette de ce que l'on voudrait obtenir :



Figure 3: Maquette de l'application

1: Barre de menus

2: Vue graphique pour afficher le graphe

3: Vue textuelle pour afficher le fichier DOT

## 4. Digramme de use case

(à discuter sur ce diagramme, il faudra surement le changer, peut-être faire des use case sur les vues, peut-être mettre des "extends" pour créer-éditer et créer-supprimer).

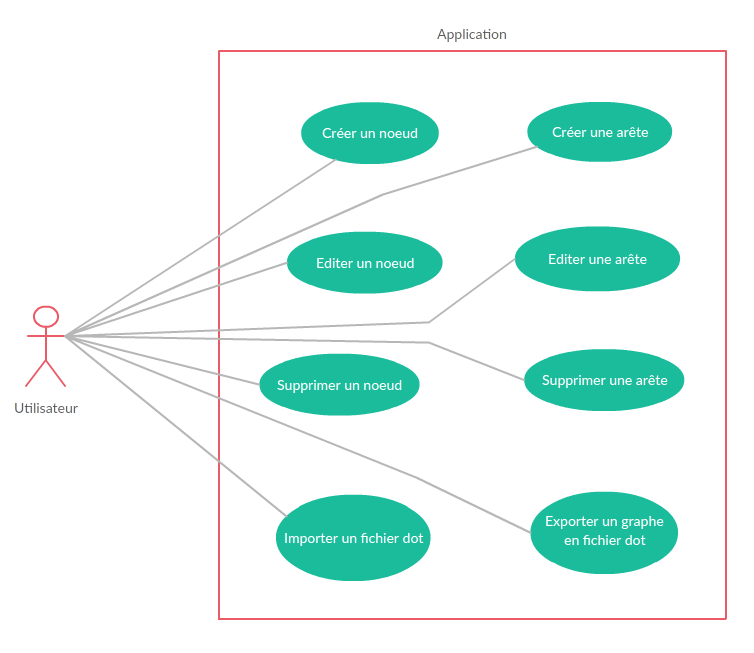


Figure 4: Diagramme de use case

Remarque:

Lorsque l'on parle d'éditer un nœud ou une arête, il s'agit de l'édition d'un de ses attributs (par exemple son label).

## 5. Outils choisis et pourquoi

(Insérer des images peut-être ?)

Tout d'abord, nous avons choisis le langage Python pour développer notre application. Nous avons choisis ce langage pour plusieurs raisons :

* la portabilité : comme Python est multiplateforme, il est facile de distribuer l'application sur les différents plateformes.
* la gestion de la mémoire : la désallocation d'une ressource est automatiquement gérée par Python, il n'est donc pas nécessaire de se soucier de la libération des ressources ce qui permet d'éviter les fuites de mémoires et d'avoir une application robuste.
* la documentation : Python est très bien documentée, il est facile de trouver ce que l'on cherche.
* la bibliothèque standard : elle est très vaste ce qui permet de ne pas réinventer la roue et d'avoir une application robuste.
* PEP (Python Enhancement Proposal) : nous en parlerons plus tard
* c'est un langage orienté objet (peut-être qu'il faut le retirer ou argumenter ?)

En ce qui concerne la bibliothèque graphique, nous avons décidé d'utiliser PyQt (qui est la version Python de Qt). PyQt a été développée dans le but de créer des interfaces utilisateurs et c'est ce qui nous intéresse dans notre cas. De plus elle est très bien documentée et est multiplateforme. C'est également une des seules (voire la seule) bibliothèque graphique à disposer d'une scène pour manipuler des objets graphiques. Il n'est donc pas nécessaire de recoder toute la partie qui permet de gérer l'ajout, le déplacement ou la surpression d'un élément graphique de la scène.

Ensuite, nous avons utilisé Git pour partager facilement le code source, avoir un backup en cas de problème et avoir un contrôle de version.

Enfin nous avons utilisé l'IDE LiClipse qui est une version customisée d'Eclipse. Il n'est pas primordial d'utiliser un IDE pour développer mais LiClipse apporte différents avantages :

* le plugin PyDev qui permet de développer en Python.
* le plugin EGit qui permet d'utiliser Git directement dans LiClipse via l'interface graphique.
* un débogueur pour analyser les bugs du programme.
* l'auto-complétion qui permet de gagner en rapidité.

II - Développement d'une application avec une interface graphique

Comme le but de notre projet était de développer une application avec une interface graphique, nous avons dû réfléchir à la façon dont nous allions organiser notre projet pour qu'il soit maintenable, lisible et évolutif (dans le sens où il est facile d'ajouter des extensions).

# 1. Architecture choisie: MVC

## 1.1 Présentation

Nous avons choisi d'utiliser une architecture MVC (Model-View-Controller). Cette architecture permet de séparer le code source en 3 différents composants:

* Modèle: Le modèle représente l'ensemble des données sur lesquelles on veut travailler. Il doit offrir des méthodes pour pouvoir mettre à jour (ajout/édition/suppression par exemple) les données ou bien tout simplement les récupérer.
* Vue: La vue est le résultat du modèle, c'est à dire qu'il représente les données du modèle sous forme textuelle ou graphique, c'est ce que voit l'utilisateur à l'écran. Mis à part afficher le résultat du modèle, la vue est également censée recevoir les actions extérieures qui interagissent avec l'interface (comme les clics souris, la saisie d'un texte, etc.).
* Contrôleur: Le contrôleur permet au modèle et à la vue de communiquer entre eux. Il faut savoir que dans l'architecture MVC, chaque composant est "indépendant" de l'autre. La vue signale au contrôleur qu'une mise à jour doit avoir lieu qui lui même signale le modèle. Le modèle effectue alors sa mise à jour puis notifie les contrôleurs qui vont à leur tour notifier les différentes vues. Les différents vues n'auront plus qu'à mettre à jour le résultat.

L'avantage de cette architecture est l'indépendance de chaque composant. Cette séparation du code source permet d'avoir un projet plus maintenable. Une approche naïve aurait été de développer l'application en fusionnant le modèle et la vue. Cependant cette solution pose un problème majeur. Imaginons qu'un jour on décide de changer de bibliothèque graphique, comme les deux composants aurait été lié il aurait été très difficile de faire toutes les modifications de code pour tout réadapter avec cette nouvelle bibliothèque sans engendrer de bug. Cette séparation permet donc de pourvoir simplement remplacer la vue ou même avoir plusieurs vues pour un seul modèle: c'est exactement ce que nous voulons puisque nous avons besoin d'une vue graphique et d'une vue textuelle pour manipuler notre graphe.

## 1.2 Mécanisme mis en place

Pour la mise en place d'une architecture MVC, il existe plusieurs façons de mettre en relation les différents composants. Nous allons donc ici présenter la manière dont nous avons mis en relation nos trois différents composants.

(screenshot diagramme de classes mvc avec observer/subject).

Comme on peut le voir sur le digramme ci-dessus, le Modèle hérite de la classe Subject et peut posséder plusieurs Contrôleurs qui sont des Observer. Un Contrôleur a une référence vers le Modèle et une Vue vu qu'il sert de communication entre ces deux composants. Une Vue n'a qu'une seul référence sur le Contrôleur.

Scénario de communication:

Prenons le cas de mis à jour de la valeur d'un texte. L'utilisateur va rentrer un texte (sur la Vue). Lors de la confirmation, la Vue va appeler la méthode onEditText(str) de son Contrôleur qui a son tour va appeler la méthode editText(str) de son Modèle. Le Modèle va pouvoir mettre à jour le texte et une fois qu'il a terminé il va appeler la méthode notify() qui consiste à appeler la méthode update() de chacun de ses Observer (donc de ses Contrôleurs). Chacun de ces Contrôleurs vont ensuite appeler la méthode update(str) de leur Vue qui vont ensuite mettre à jour leur interface.

(Schéma de communication).

# 2. Architecture du projet

## 2.1 Organisation du projet

Le projet est organisé en différents packages afin de regrouper les éléments qui ont la même logique ensemble. Voici l'organisation des packages du projet:

* app: contient "le moteur" de l'application
* controller: ensemble des différents contrôleurs
* enumeration: ensemble des différentes énumération
* model: ressources qui concerne le Modèle
* observer: contient ce qui concerne Observer/Subject pour le MVC
* utils: utilitaires
* view: ensembles des différentes vues

Cette structure permet de trouver facilement ce que l'on cherche et il est assez trivial de savoir ou placer les nouveaux fichiers en cas d'extension.

(Insérer diagramme de composants ?).

## 2.2. Digramme de classe des principaux composants

(mettre à jour diagramme addWidget dans MainWindow?)

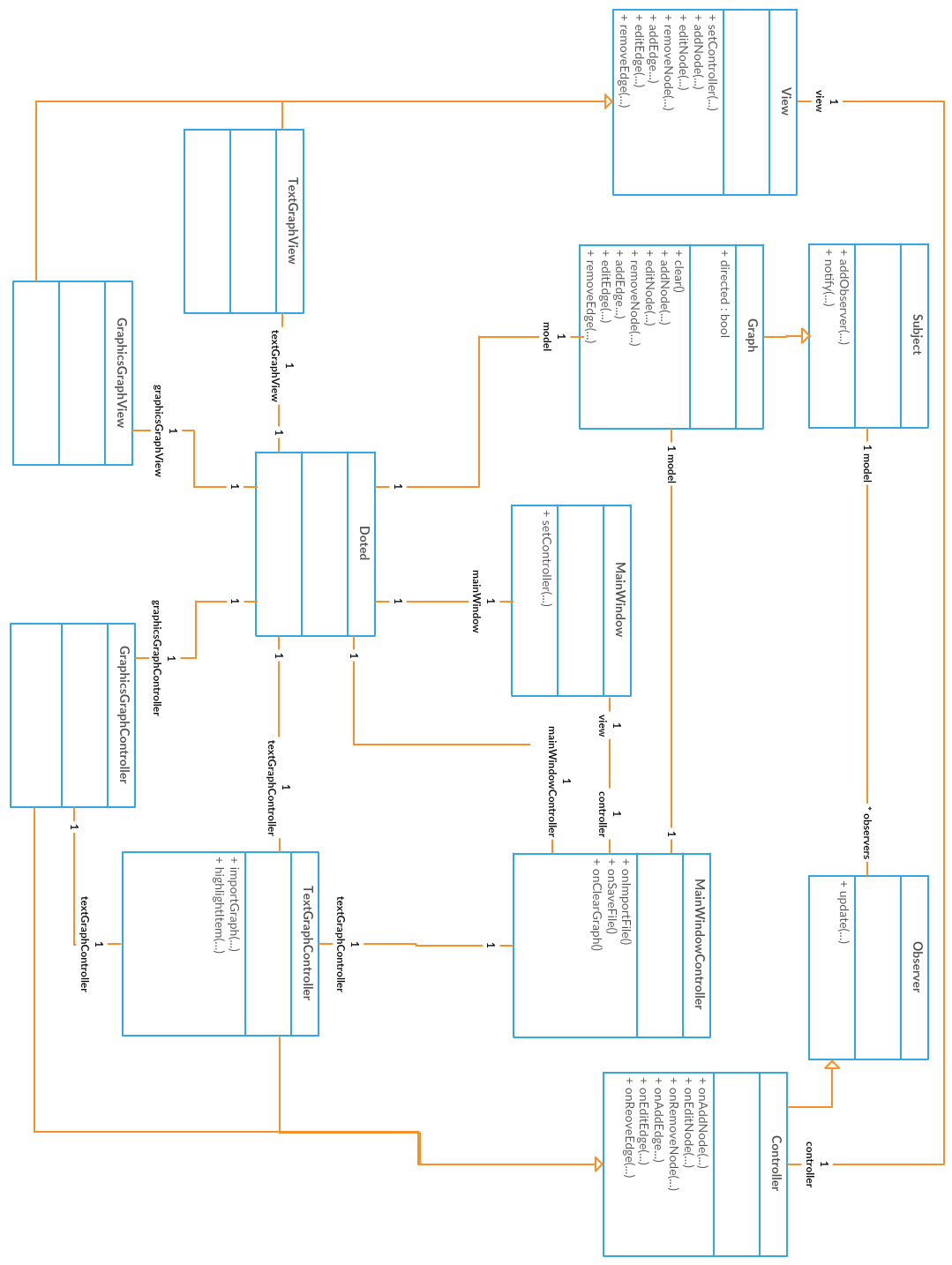


Figure : Semi-diagramme de classes

Le digramme de classes ci-dessus n'est qu'un semi-diagramme de classes de l'application dotEd. En effet seul les éléments important ont été mis sur le diagramme de classe c'est à dire qu'il ne contient pas toutes les classes, tous les attributs et toutes les méthodes. De plus les "..." dans les parenthèses signifient que ces méthodes ont des arguments mais ils n'ont pas été spécifiés.

La classe Doted est un peu le cœur de l'application. C'est elle qui va s'occuper d'instancier le modèle, les contrôleurs ainsi que toutes les vues.

Comme on peut le voir sur le diagramme de classes, nous avons deux couples MVC (partageant le même modèle Graph):

* Graph-GraphicsGraphView-GraphicsGraphController: pour la vue graphique.
* Graph-TextGraphView-TextGraphController: pour la vue textuelle.

Nous avons également le MVC Graph-MainWindow-MainWindowController qui est différent des deux autres couples MVC. En effet la classe MainWindow représente en fait la vue principale de l'application, c'est à dire que c'est elle qui va afficher les 2 différentes vues, la barre de menus, les différents boutons, etc. La vue MainWindow ne rend pas d'affichage du modèle Graph. C'est donc pour cela qu'elle n'hérite pas de la classe View comme elle n'as pas besoin de mettre à jour sa vue en cas de modification du modèle. Pour la même raison la classe MainWindowController n'hérite donc pas de la classe Controller. Nous avons tout de même besoin d'avoir une classe MainWindowController au cas où nous décidons de changer la vue principale MainWindow avec une autre bibliothèque graphique.

Remarque 1:

La classe View ne possède pas de méthode "update" à proprement parler. En effet la méthode update est dispatchée en plusieurs méthodes add(Node/Edge), edit(Node/Edge), remove(Node/Edge). C'est au niveau du modèle qu'est choisi quelle méthode update sera appelée. Le modèle envoie l'information à son contrôleur qui lui va savoir quelle méthode update appeler.

Remarque 2:

Il faut noter qu'il existe une classe Node et une classe Edge qui sont toutes les deux utilisées par la classe Graph pour modéliser les nœuds et les arêtes.

(expliquer pourquoi il y a plusieurs contrôleurs et pourquoi il y a des liaisons inter-contrôleurs)

# 3. Interfaces graphiques

Dans cette partie nous allons nous intéressés à la gestion des différentes vues de l'application. Pour rappel, nous avons utilisé la bibliothèque graphique PyQt. Ainsi quand nous parlerons d'une classe QClassName, cela signifie que l'on parle d'une classe de PyQt.

## 3.1 La vue principale: MainWindow

Voici un aperçu de l'application:

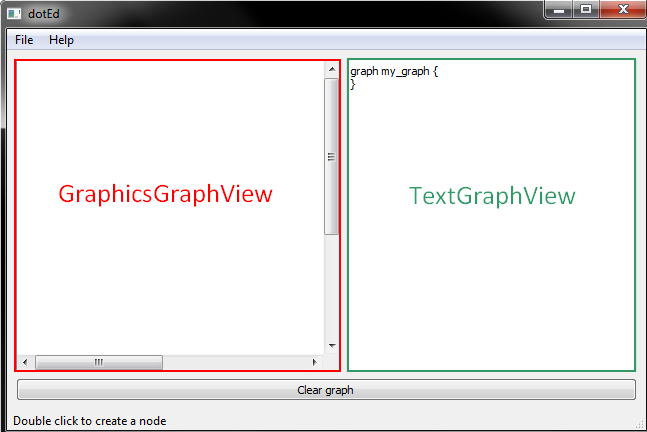


Figure : Aperçu de l'application

La vue de l'application est générée grâce à la classe MainWindow. Comme on peut le voir sur l'image ci-dessus, elle contient nos deux vues principales GraphicsGraphView et TextGraphView. Ces deux vues sont séparées par une barre de séparation (classe QSplitter) qui permet d'agrandir ou de rétrécir une vue ou l'autre. Elle contient également une barre de menus qui permet d'effectuer différentes actions comme importer ou exporter un fichier DOT. Le bouton "Clear Graph" permet juste d'effacer le graphe dans les deux vues.

La vue MainWindow est modulaire, c'est à dire qu'il est facile d'y ajouter de nouvelles vues si nous souhaitons un jour en rajouter. Elle dispose en effet d'une méthode addWidget(...) qui permet d'ajouter une vue. Toutes les vues que l'on voudra ajouter devront hériter de la classe View et d'une classe Qt qui hérite de la classe QWidget.

## 3.2 La vue graphique: GraphicsGraphView

La vue graphique est celle qui est encadrée en rouge sur la figure 6. C'est cette vue qui va s'occuper d'afficher le graphe (donc les nœuds et les arêtes). Elle permet donc d'ajouter/éditer/supprimer des nœuds/arêtes.

### 3.2.1 Description de la classe GraphicsGraphView en détail

Voici comment semi-diagramme de classe de GraphicsGraphView en détail:

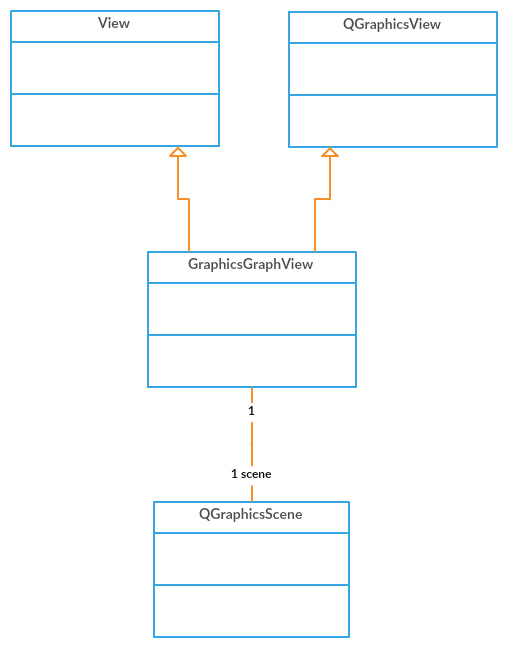


Figure : Semi-diagramme de classe de GraphicsGraphView en détail

La classe principale ici est GraphicsGraphView. Elle hérite de la classe View et de la classe QGraphicsView (qui hérite de la classe QWidget). Elle respecte donc les critères pour pouvoir être ajoutée à MainWindow avec la méthode addWidget(...). Comme on peut le voir sur le diagramme ci-dessus, la GraphicsGraphView dispose d'une QGraphicsScene. C'est cette scène qui s'occupe de l'affichage des éléments graphiques et qui nous permet de les manipuler. Cette QGraphicsScene dispose de plusieurs évènements (comme le clic souris, le déplacement de la souris, etc.). On peut donc redéfinir ces évènements afin d'effectuer les actions que l'on souhaite.

### 3.2.2 La gestion des nœuds

Nous allons ici nous intéressés au lien des nœuds graphiques avec la classe GraphicsGraphView.

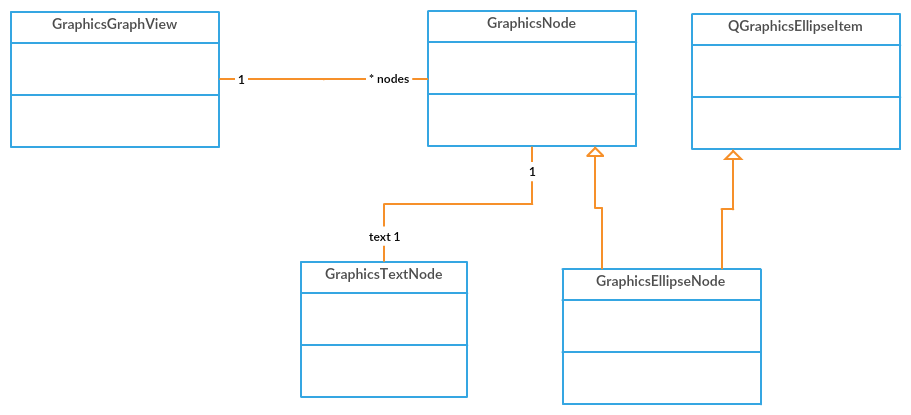


Figure : Semi-diagramme de classe de GraphicsGraphView avec nœuds

Pour ce qui est de la gestion des nœuds graphiques, nous avons créé une classe mère GraphicsNode qui regroupe les attributs et les méthodes communes à tous les nœuds graphiques que l'on pourra créer. Chaque GraphicsNode dispose d'une GraphicsTextNode qui permet simplement d'afficher le texte (attribut label de DOT) du nœud. Pour le moment, l'application ne gère que la forme ellipse d'où la classe GraphicsEllipseNode. Lorsque que l'on veut ajouter une nouvelle forme, il faut que la nouvelle classe hérite de la classe GraphicsNode ainsi que d'une classe PyQt de type QAbstractGraphicsShapeItem. En l'occurrence, il n'existe que les formes polygone, ellipse et rectangle qui ont déjà été implémentées par PyQt. Si l'on souhaite ajouter une forme étoile par exemple, il faudra la programmer soi-même.

### 3.2.3 La gestion des arêtes

Nous allons ici nous intéressés au lien des arêtes graphiques avec la classe GraphicsGraphView.

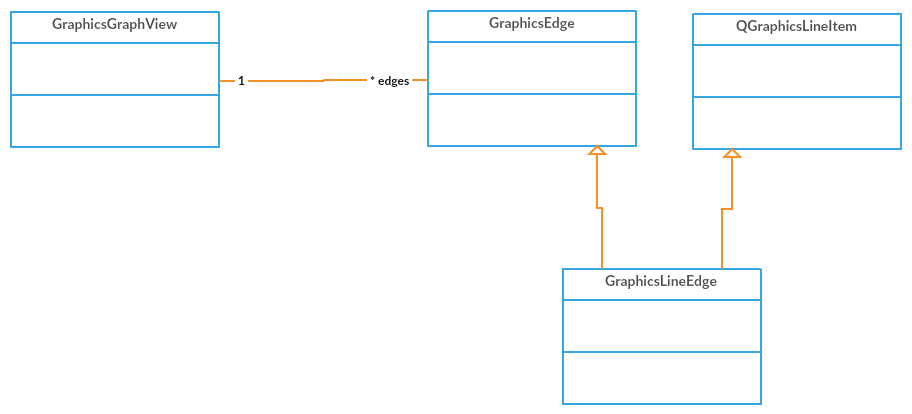


Figure : Semi-diagramme de classe de GraphicsGraphView avec arêtes

Le mécanisme de gestion des arêtes ressemble fortement à celui des nœuds. Tout comme pour les nœuds graphiques, les arêtes dispose d'une classe mère GraphicsEdge qui regroupe les attributs et méthodes communes à toutes les arêtes graphiques que l'on voudra créer. Seules les lignes linéaires (donc en ligne droite, la classe QGraphicsLineItem) sont gérées pour le moment. Lorsque l'on souhaite ajouter une nouvelle forme d'arrête, il faut la faire hériter de GraphicsEdge et également d'une classe PyQt de type QGraphicsItem. Cependant nous avons constater que seules les lignes linéaires ont déjà été implantées par PyQt. Si l'on veut avoir une ligne courbée, il faudra programmer soit même cet élément graphique.

### 3.2.4 La gestion des évènements

## 3.3 La vue graphique: TextGraphView

IV - Maintenabilité et distribution

## 1. Exemple d'extension

## 2. PEP, licence, packaging

PEP : c'est un document de conception qui décrit l'ensemble des conventions et bonnes pratiques pour coder en Python. En respectant ces normes, le code sera plus maintenable et une nouvelle personne qui souhaite travailler sur ce projet aura beaucoup plus de facilité à le comprendre.

Licence de base : GNU GPL

Peut-être prendre la licence MIT qui est totalement libre ?

Packaging :

Conclusion

Ce projet nous a permis d'apprendre énormément de choses. Tout d'abord c'était la première fois que nous utilisions une architecture MVC pour faire une application et nous avons donc dû nous renseigner sur ses différents mécanismes et avantages. Nous ne connaissions pas le langage Python et nous avons donc dû l'apprendre pour pouvoir développer l'application. Comme vous avez pu le voir dans la dernière partie du rapport, nous avons fait notre possible pour suivre la philosophie de Python en essayant de respecter les directives du document de conception PEP. De plus comme c'est un "gros" projet qui a pour but d'être maintenu et être évolutif, il a fallut penser différemment. En effet il fallait faire en sorte d'avoir une application "générique" dans le sens où l'ajout d'une nouvel attribut DOT ou une nouvelle vue se ferait de manière simple. Dans nos anciens projets nous n'avions pas réellement eu besoin de réfléchir à tous ces problèmes.

Tables des illustrations

[Figure 1: Grammaire du langage DOT 4](#_Toc450676123)

[Figure 2: Représentation graphique d'un fichier DOT 5](#_Toc450676124)

[Figure 3: Maquette de l'application 6](#_Toc450676125)

[Figure 4: Diagramme de use case 7](#_Toc450676126)

[Figure 5: Semi-diagramme de classes 12](file:///C:\Users\Victor\git\dotEd\Rapport.docx#_Toc450676127)

[Figure 6: Aperçu de l'application 14](#_Toc450676128)

[Figure 7: Semi-diagramme de classe de GraphicsGraphView en détail 15](#_Toc450676129)

[Figure 8: Semi-diagramme de classe de GraphicsGraphView avec nœuds 16](#_Toc450676130)

[Figure 9: Semi-diagramme de classe de GraphicsGraphView avec arêtes 17](#_Toc450676131)

Annexes

(Peut-être que cette partie est inutile)