

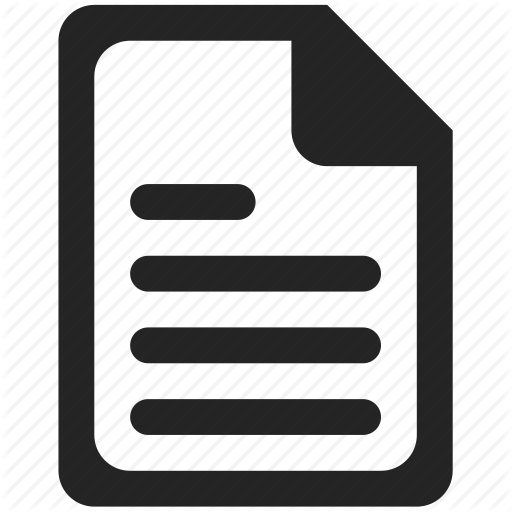
**Rapport Projet LO21**

**Conception et Programmation orientée objet**

Projet PluriNote

**CAUDRELIER Thomas**

**SUTKOWSKI Quentin**



Le 15/06/17

**Semestre P17**

Table des matières

[Introduction 3](#_Toc485309291)

[I- Organisation 3](#_Toc485309292)

[II- Choix d’architecture 4](#_Toc485309293)

[Explication des concepts généraux 4](#_Toc485309294)

[Les Design Patterns 4](#_Toc485309295)

[Le singleton 4](#_Toc485309296)

[L’iterator 4](#_Toc485309297)

[Le strategy 5](#_Toc485309298)

[L’observer 6](#_Toc485309299)

[Le factory 6](#_Toc485309300)

[III- Choix de spécifications et techniques 6](#_Toc485309301)

[Le choix sur les relations 7](#_Toc485309302)

[Le choix des onglets 7](#_Toc485309303)

[Le choix du XML 8](#_Toc485309304)

[La méthode d’implémentation pour les arborescences 8](#_Toc485309305)

[IV- Améliorations et perspectives 10](#_Toc485309306)

[Utilisation d’un patron Decorator 10](#_Toc485309307)

[Sauvegarde du contexte 10](#_Toc485309308)

[Améliorer l’objectif MVC 11](#_Toc485309309)

[Annuler/Rétablir 11](#_Toc485309310)

[Amélioration graphique 11](#_Toc485309311)

[Conclusion 11](#_Toc485309312)

[Annexes 12](#_Toc485309313)

[UML 12](#_Toc485309314)

[Diagramme d’utilisation 13](#_Toc485309315)

[Backlog 13](#_Toc485309316)

[Table des illustrations 14](#_Toc485309317)

# Introduction

Ce projet, dans le cadre de l’UV LO21, a pour objectif de concevoir et développer une application « *PluriNotes* ». Cette application vise à gérer un ensemble de notes pouvant prendre différentes formes : articles, tâches, images et vidéos, par exemple.

Nous avions à notre disposition diverses spécifications à respecter. Notre objectif principal a donc été de répondre à un maximum de ces demandes tout en respectant les contraintes.

Dans la suite de ce rapport, nous allons exposer notre mode d’organisation global, nous présenterons également nos différents choix d’architecture et d’implémentation. Nous justifierons aussi certaines limites de notre application, qui en effet reste améliorable.

On retrouvera également en annexe l’ensemble des documents permettant de détailler les spécificités de notre rendu. Enfin, une vidéo d’utilisation de notre application est accessible ici : [**URL YOUTUBE[[1]](#footnote-1)**](https://www.youtube.com/watch?v=yBJGABFjtcg&feature=youtu.be)**.**

# Organisation

Etant en binôme, nous avons décidé de nous organiser de la manière la plus précise possible afin d’aller aussi loin dans le projet que les groupes de 3. Ainsi, dès le début, nous nous sommes mis d’accord sur divers points.

Nous avons d’abord passé les premières semaines sur la clarification des spécifications, puis aussi et surtout sur la conception de notre architecture. Nous avons d’abord construit un modèle simplifié, puis nous avons itéré le processus en ajoutant à chaque fois une couche de complexité, pour à terme répondre à toutes les contraintes.

Nous avons essayé de construire notre architecture[[2]](#footnote-2) de la manière la plus évolutive possible : passage par des classes abstraites, mise en avant de relation de composition, et également bien sûr l’utilisation de design pattern.

En fil rouge et afin de permettre l’évolution en parallèle de notre projet, nous avons mis en place un BackLog [[3]](#footnote-3)décomposant nos objectifs en taches. Ces taches ont été ensuite gérées via un dépôt GIT. Egalement, nous nous sommes retrouvé chaque semaine afin de faire des points sur le projet, fixer les échéances suivantes et s’entraidant sur diverses problématiques techniques.

Pour l’aspect technique et choix logiciels nous avons utilisé :

* *Modelio* pour la modelisation UML et diagramme d’utilisation
* *Cacoo.com* pour le maquettage graphique
* *Git* pour le gestionnaire de version
* *Qt Creator* pour l’ensemble du code
* *Google Drive* pour le partage de divers fichiers dont le BackLog

# Choix architecturaux

## Explication des concepts généraux

Afin de rendre la plus modulable possible l’application, nous avons utilisé dès que possible l’héritage. Par exemple, nous avons construit les classes *AbstractNote* et *Media*. Ces classes abstraites permettent de centraliser les attributs et méthodes des différents types de notes.

Nous avons également utilisé dès que possible le polymorphisme afin de simplifier le modèle et de gagner en modularité.

Egalement, vous remarquerez qu’il existe un nombre relativement conséquents de classes. La conception a été conçu volontairement dans une envie « d’over-design », une des contraintes étant de rendre le projet le plus modulable possible.

## Les Design Patterns

Nous avons mis en place différents design pattern (patron de conception) dans ce projet. Ces méthodes sont reconnues comme « bonnes pratiques ». Ceux-ci permettent entre autres de rendre le logiciel plus pérenne, en permettant une plus grande évolutivité et facilité de compréhension. Nous les présenterons succinctement par la suite.

### Le singleton

Ce singleton permet d’assurer un contrôle sur le nombre d’instances que peut retourner une classe. Ce contrôle passe par différents moyens : déclarer privés les constructeurs de la classe, instancier de manière unique l’objet *static* de la classe via un pseudo constructeur.

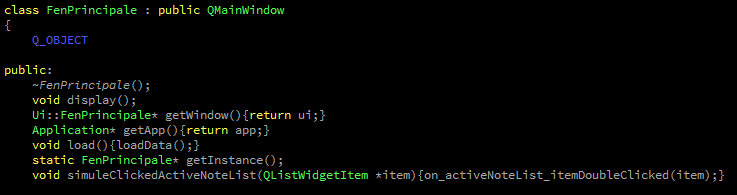


Figure 1 - Vue partielle de la classe FenPrincipale. Pseudo constructeur static visible



Figure 2 - Vue partielle de la classe FenPrincipale. Constructeur privé visible

Nous avons implémenté le singleton sur la classe *FenPrincipale* gérant les interactions User Interface / Code. On retrouve les éléments contrôle décrient précédemment : constructeur en privé et pseudo constructeur *static*. Ce design pattern participe à la protection de notre l’architecture : on empêche en effet d’avoir plusieurs entités de cette classe, ce qui serait absurde pour cette interface centrale qu’est *FenPrincipale*.

### L’iterator

Il s’agit du plus commun des design patterns gérant le comportement. Il permet de limiter la vision d’une collection pour l’utilisateur, et de parcourir tous les éléments contenus dans cette collection. Il ajoute un niveau de contrôle par rapport à un parcours classique, car il isole la structure interne du conteneur (parfois complexe) du parcours lui-même.

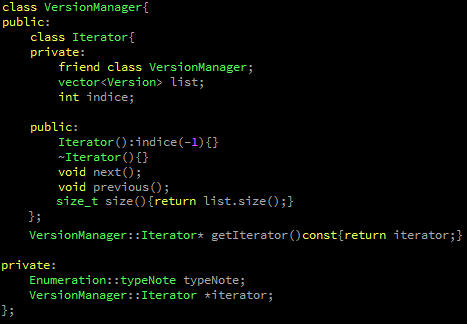


Figure 3- Vue partielle de la classe VersionManager - Encapsulation de l'iterateur

Nous utilisons ce design pattern dans la classe *VersionManager* de parcourir l’ensemble de la liste des Versions pour une note donnée. Nous pouvons ainsi parcourir les versions indépendamment de leur structure interne relativement complexe.

### Le strategy

Ce patron de conception gère également le comportement. Il permet de permuter dynamiquement entre différents algorithmes. Il encapsule divers algorithmes ayant une fonction similaire.



Figure 4 - UML du patron Stategy[[4]](#footnote-4)

Il nous est utile dans la gestion *XML* de nos données, dans le sens où celui-ci permet facilement d’analyser des données, afin de les stocker mais également de les lire. Il participe au caractère évolutif de l’application, car il permet notamment de passer facilement d’un système de gestion de type base de données à un système *XML* ou *JSon*.

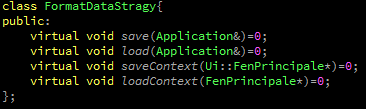


Figure 5 - Exemple de stratégie dans notre classe FormatDataStrategy

Nous voyons ici l’intérêt du design pattern : nos méthodes génériques permettront la gestion uniforme des méthodes de gestion de nos données. Si plus tard nous voulions passer sur une gestion SGBD, il suffira de mettre en place la nouvelle classe dans la « famille » de l’interface FormatDataStrategy. Chaque classe stratégie concrète à la charge de la gestion du client de gestion associé.

### L’observer

Cet autre design pattern permet de faire le lien entre un modèle et un objet observé (souvent une vue).

Dans notre cas, nous nous sommes appropriés ce patron pour les classes *Notes* et *Relations*. Les observer associés ont pour rôle de faire la liaison entre le modèles (par exemple la classe Relation et plus particulièrement le Manager de Relation) et la vue (par exemple *FenPrincipale*).

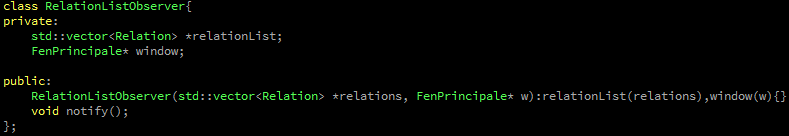


Figure 6 - Exemple d'observeur pour la gestion des relations

### Le factory

Afin de séparer la création des différents types d’objets, de leur utilisation, le patron fabrique (factory) nous a paru primordiale. Cela permet alors par la suite de faciliter la création de nouveaux objets implémentant AbstractNote.

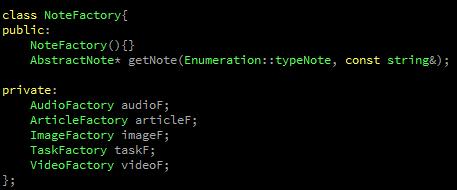


Figure 7 - Classe NoteFactory permettant l'instanciation des notes diverses

Dans notre cas, nous utilisons le patron *factory* sur *NoteFactory.* Cette classe permet d’instancier les différentes notes de manière centralisée, en spécialisant la production selon le type du « modèle » a créé (Tâche, article, …). Elle appelle les différentes fabriques concrètes ensuite qui utiliseront la méthode *getNote*. Si par la suite on décide d’implémenter un nouveau type de note, il suffira d’ajouter un créateur concret (à l’image de *TaskFactory*) pour créer un nouveau type d’objet, sans avoir à reprendre tous le processus de construction.

### Le memento

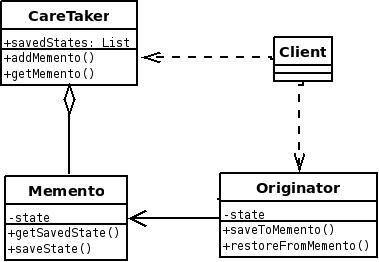
Ce design pattern nous a permis de gérer les différentes versions de note. Il permet en effet de restaurer un état précédent d’un objet, sans violer le principe d’encapsulation.

Figure 8- Memento

# Choix de spécifications et techniques

## Le choix sur les relations

En l’absence de précision sur le comportement d’une relation lorsque tous les couples la composant sont supprimés, nous avons dû trancher sur la suppression ou non de la dite-relation.

Apres réflexion, nous avons décidé de ne pas supprimer la relation dans ce cas. En effet, en faisant cela, nous permettons à l’utilisateur de revenir sur cette relation et de lui ajouter de nouveaux couples, sans avoir besoin de la recréer totalement. Toutefois, si l’utilisateur souhaite vraiment supprimer la relation, alors il peut le faire manuellement via la suppression de relation classique.

## Le choix des onglets

Pour le côté User Interface, nous avons de découper les différents aspects de l’application par des onglets. Ceci nous ont semblé adapté aux spécifications, car permettent de séparés les fonctionnalités (gestion des notes, gestion des relations, gestion de la corbeille et enfin des paramètres). L’expérience utilisateur nous semble alors être plus intuitive. On évite aussi en cela de passer par une multitude de fenêtre qui pourrait perdre l’utilisateur.

D’un point de vue technique, ceci nous a également permis d’avoir accès de manière centralisée à tous nos objets graphiques.

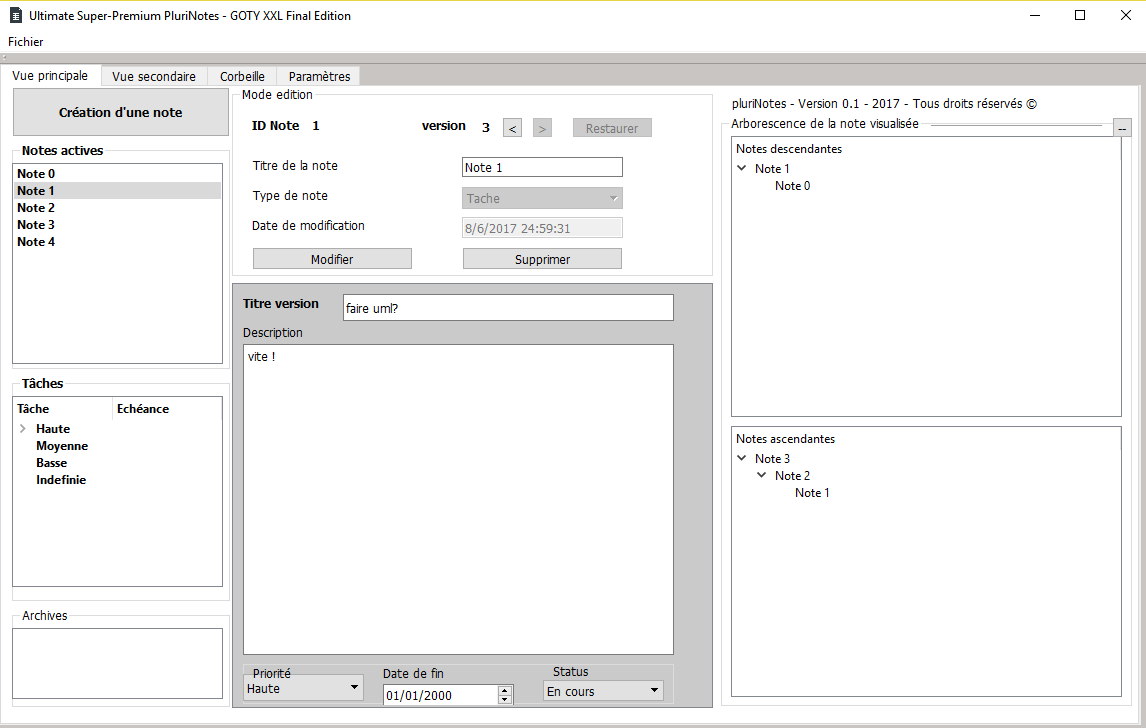


Figure 9 - Exemple de vue de l'onglet principale

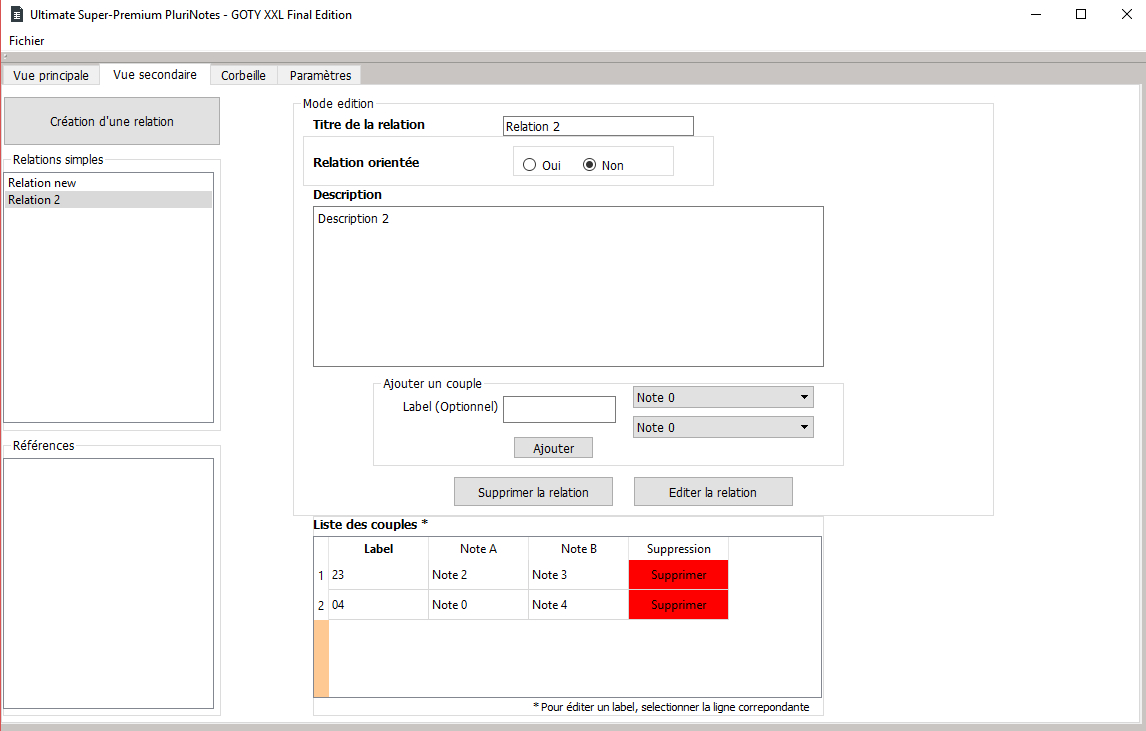


Figure 10 - Exemple de vue de l'onglet secondaire

## Le choix du XML

Nous avions une contrainte de persistance des données. Etant libres de choisir le format de stockage, nous avons décidé de partir sur une solution de gestion *XML*. Cette solution nous a paru adapter car plus simple et rapide à mettre en place. En effet, Qt possède entre autres des méthodes de gestions pour le XML. En implémentant le patron de conception *Strategy* pour les classes gérant le XML, nous avons complété ces méthodes.

## La méthode d’implémentation pour les arborescences

Pour l’affichage des arborescences ascendantes et descandantes, nous avons implémenté notre algorithme de façon récursive. Ce choix algorithmique nous a semblé intéressant à présenter ici, de par sa structure, son « élégance algorithmique », et sa complexité faible (linéaire, O(n)) comparativement au parcours nécessaire.

Nous présenterons ci-dessous la procédure de remplissage pour l’arborescence ascendante. Celle pour l’arborescence descendante est très proche.

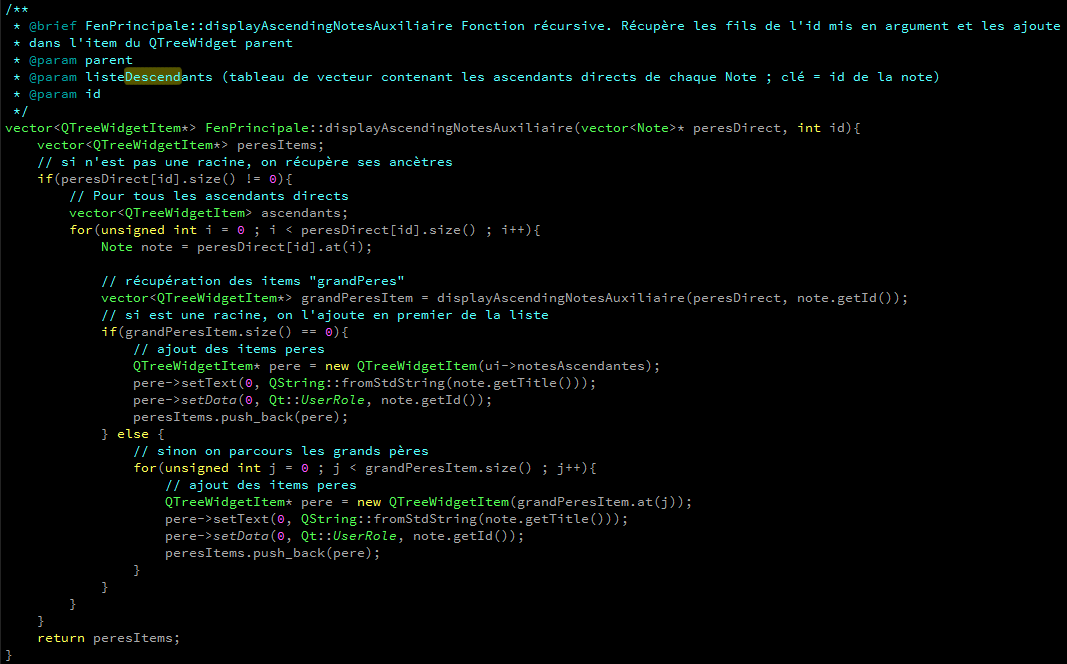


Figure 11 - Fonction displayAscendingNotesAuxiliaire. Permet l'affichage récursif des items liés aux notes ascendantes

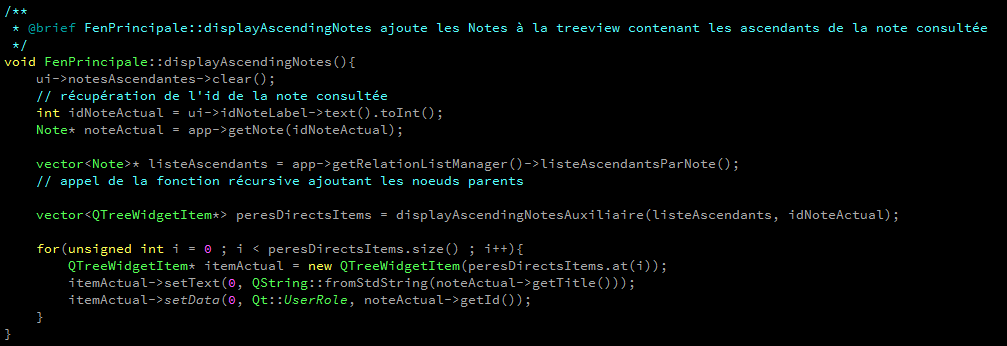


Figure 12 - Fonction displayAscendingNotes. Permet l'affichage de la note consultée + remplissage du vecteur des notes ascendantes

La fonction *DisplayAscendingNotes*, on récupère la note actuellement sélectionnée. On appelle alors la fonction *listAscendantsParNote*, qui permet de récupérer un vecteur contenant toutes les notes ascendantes, dont les index sont fixés aux id des notes ascendantes.

Enfin, on appelle la fonction *displayAscendingNotesAuxiliaire* qui va remonter de façon récursive l’arborescence, créer les items liés aux notes parcourues, et les afficher. Certaines vérifications nous permettent de déterminer si la note parcourue est une racine, ou si le parcours doit continuer.

# Informations supplémentaires

Ci-dessous les informations importantes quant à l’usage de notre application :

* Afin de sauvegarder les données, vous devrez créer un dossier XML ainsi que les fichiers suivants :
  + – contexte.xml
  + – notes.xml
  + – parameters.xml
  + – relations.xml

Si ces fichiers ne sont pas créés, le logiciel vous informera de ce problème.

# Améliorations et perspectives

Il existe en effet des limites et des améliorations potentielles pour ce projet. En effet, nous disposons de trois contraintes pour répondre aux diverses spécifications : des contraintes de temps, des contraintes techniques (nous ne connaissions pas Qt en détail au début du projet), ainsi que des contraintes de qualité. Le but n’étant en effet pas de répondre uniquement aux spécifications, mais de permettre une architecture de qualité, évolutive, modulaire et maintenable.

Egalement il existe des améliorations que nous aurions pu développer (certains patrons de conception en l’occurrence), mais nous étions déjà dans une phase trop avancée de programmation (il ne nous était donc pas possible de faire une refonte), et il était difficile de prévoir tous les problèmes en amont lors de la phase de conception.

## Utilisation d’un patron Decorator

Afin de tirer profit de la factorisation de l’architecture, nous aurions pu, afin d’éviter de manière systématique l’héritage (qui complexifie rapidement le modèle), mettre en place le patron de conception Décorateur (decorator). Celui-ci permettant d’ajouter des fonctionnalités nouvelles à une classe de façon dynamique, sans impacter les classes qui l’utilisent ou en héritent.

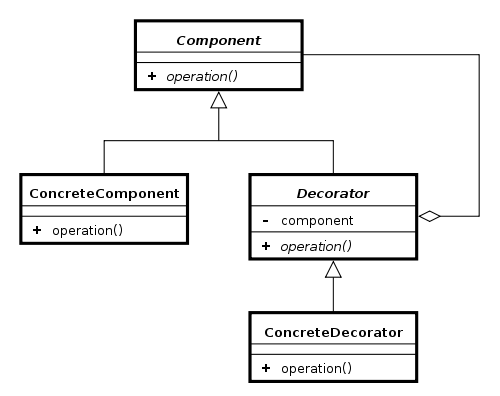
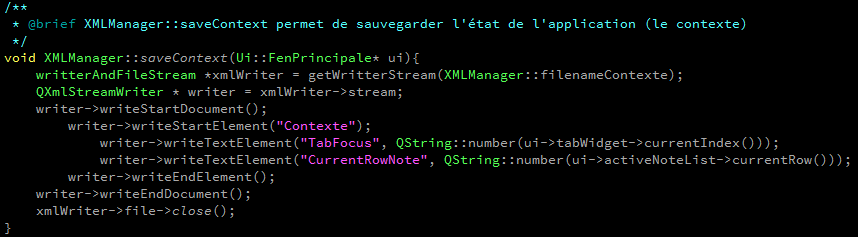


Figure 13 - Schéma UML DP Decorator

## Sauvegarde du contexte

Afin de répondre aux contraintes de sauvegarde du contexte, nous avons mis en place des méthodes dans le *XMLManager* de sauvegarde et de chargement. Actuellement, ces méthodes nous permettent de récupérer et restituer l’onglet actif, la note consultée et les paramètres à la fermeture de l’application. Nous nous sommes concentrés sur ces trois récupérations majeures. Avec un peu plus de temps, nous pourrions poursuivre cette solution et l’exporter à d’autres éléments comme la récupération de la relation sélectionnée. Nous avons voulu cette solution modulaire et générique, ainsi le développement complet n’aurait pas demandé beaucoup de réflexion supplémentaire.



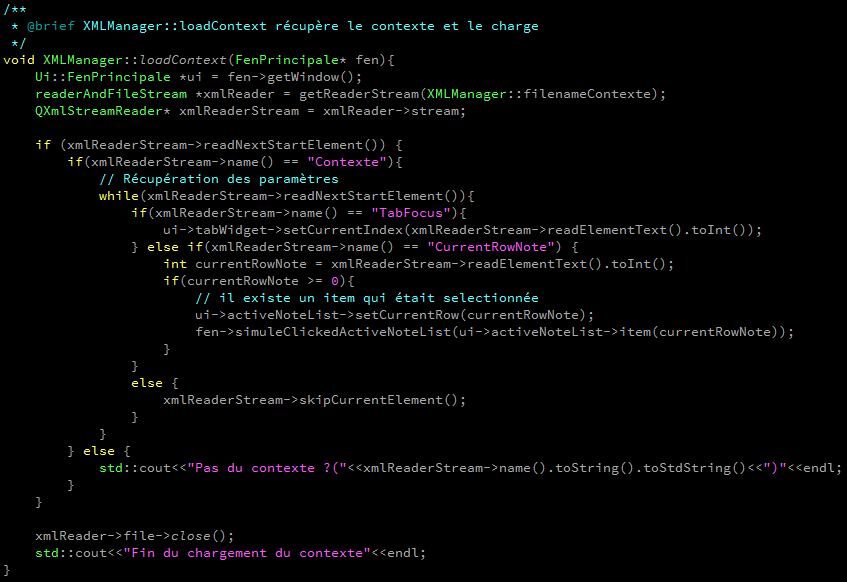


Figure 14 - Fonctions de sauvegarde du contexte

## Améliorer l’objectif MVC

Nous avons lors de nos réflexions voulu se rapprocher d’un modèle vue contrôleur, qui permet une grande clarté dans l’architecture et la simplification de gestion pour le développeur et la maintenabilité du projet. Toutefois, nous avons fait ce choix tardivement, ce qui explique que notre modèle n’est que partiellement sous la forme MVC (FenPrincipale.cpp agit ici comme un « super-controlleur », responsable de l’ensemble des évènements de l’application).

## Annuler/Rétablir

Cette fonctionnalité n’a pu être mise en place par contrainte de temps. Nous avons tout de même effectué des recherches afin de répondre à cette demande. Nous pensons que l’utilisation du *framework Qt Undo* permettrait de répondre à celle-ci de manière modulable (facilité d’ajouter de nouvelles actions sur la stack).

## Amélioration graphique

Graphiquement, notre application nous semble assez fonctionnelle et plutôt intuitive. Nous avons tenté par divers moyens, comme des labels explicatifs, de clarifier les différentes possibilités pour l’utilisateur. Une perspective de rendu responsive est possible et pourrait améliorer l’expérience utilisateur. Nous avons également essayé d’implémenter une gestion d’erreur sur diverses actions de l’utilisateur (impossibilité de créer une note sans nom, empêcher la création de deux couple A-B dans une même relation, etc). De même, nous n’avons pas eu le temps de permettre la visualiation des Médias. L’architecture a tout de même était conçu afin de permettre l’implémentation de celle-ci (Media comme classe mère de {VIDEO,AUDIO,IMAGE} avec une méthode virtuelle pure displayMedia) ;

# Conclusion

Pour finir, ce projet nous a permis de réaliser de manière complète l’application de gestion de notes *PluriNotes*. Il nous a permis de prendre en main un processus complet de mise en place logiciel : réflexion sur l’architecture, organisation humaine et technique, implémentation technique, mais également un travail de synthétisation de notre travail. Il entre donc parfaitement dans notre cursus actuel, joignant technique et relationnel.

# Annexes

## UML

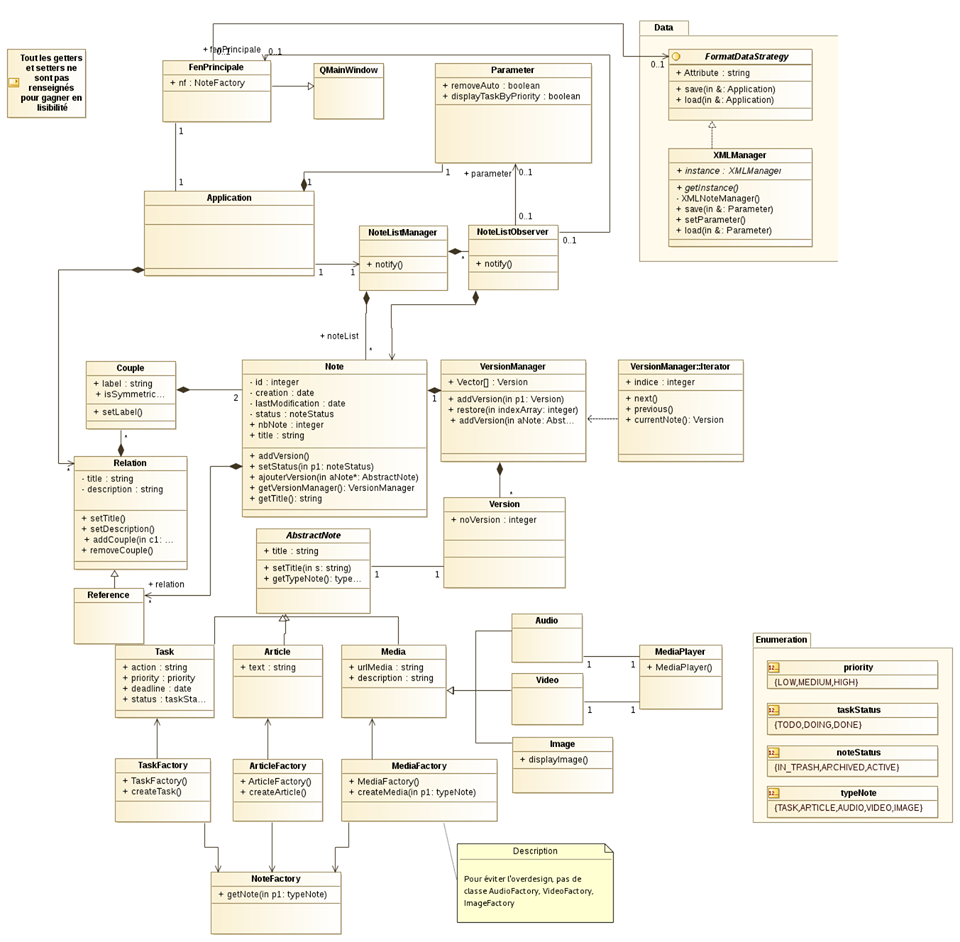
N.B : Nous avons décidé de ne pas expliciter l’ensemble de tous les attributs sur ce modèle (comme typiquement certains getters et setters) afin de le clarifier.

Figure 15 - UML de l'application pluriNotes

## Diagramme d’utilisation

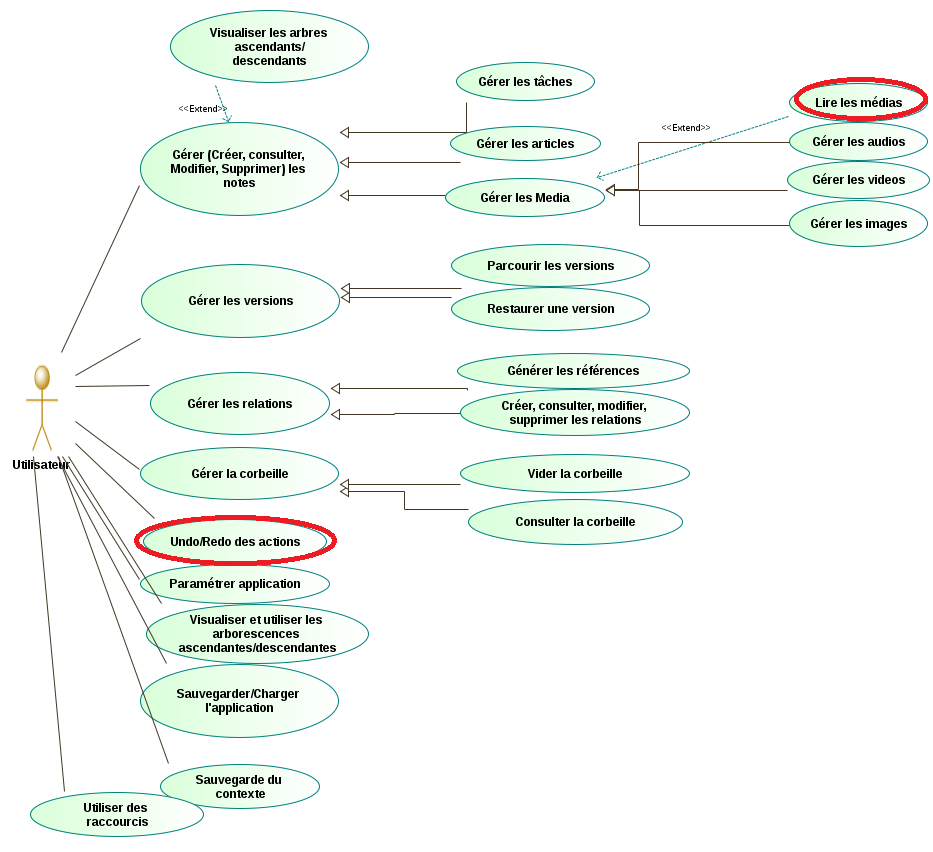


Figure 16 - Diagrammes d'utilisations de pluriNotes. En rouge : les fonctionnalités non implémentées

## Backlog

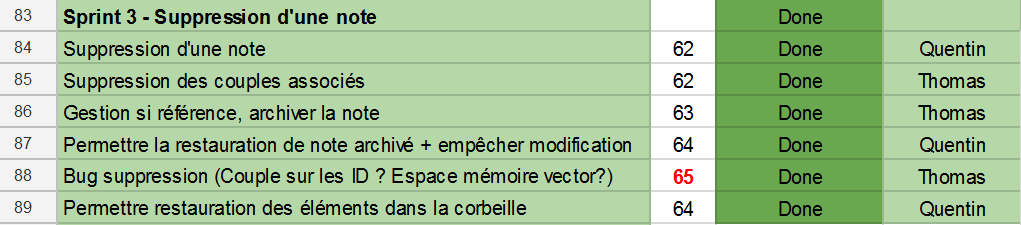


Figure 17 - Vue partielle du backlog pour le 3eme sprint

# Table des illustrations

Figure 1 - Vue partielle de la classe FenPrincipale. Pseudo constructeur static visible 4

Figure 2 - Vue partielle de la classe FenPrincipale. Constructeur privé visible 4

Figure 3- Vue partielle de la classe VersionManager - Encapsulation de l'iterateur 5

Figure 4 - UML du patron Stategy 5

Figure 5 - Exemple de stratégie dans notre classe FormatDataStrategy 5

Figure 6 - Exemple d'observeur pour la gestion des relations 6

Figure 7 - Classe NoteFactory permettant l'instanciation des notes diverses 6

Figure 8 - Exemple de vue de l'onglet principale 7

Figure 9 - Exemple de vue de l'onglet secondaire 7

Figure 10 - Fonction displayAscendingNotesAuxiliaire. Permet l'affichage récursif des items liés aux notes ascendantes 8

Figure 11 - Fonction displayAscendingNotes. Permet l'affichage de la note consultée + remplissage du vecteur des notes ascendantes 8

Figure 12 - Schéma UML DP Decorator 10

Figure 13 - Fonctions de sauvegarde du contexte 11

Figure 14 - UML de l'application pluriNotes 12

Figure 15 - Diagrammes d'utilisations de pluriNotes 13

Figure 16 - Vue partielle du backlog pour le 3eme sprint 13

1. <https://www.youtube.com/watch?v=yBJGABFjtcg&feature=youtu.be> [↑](#footnote-ref-1)
2. On retrouvera le diagramme UML final en annexe 1. [↑](#footnote-ref-2)
3. Voir annexe (extrait) [↑](#footnote-ref-3)
4. Source : Wikipedia [↑](#footnote-ref-4)