LO21 PROJET
Gestionnaire de notes : Plurinote

GARNIER Maxime · NAUDIN Louise · PEPIN Hugues

14 Juin 2017

Table des matières

1 Phase de Conception

1.1 Organisation préliminaire du travail

Afin de s'assurer d'une compréhension cohérente du sujet entre les membres du groupe, la première action a été une lecture sérieuse du sujet et la proposition d'un premier UML.

Cette formalisation a permis de poser des hypothèses sur ce qui pouvait nous sembler ambigu et de structurer le projet. Les choix faits concernant la compréhension du sujet ont été clairement indiqués dans l'UML et dans une note de clarification faite à notre intention. Ces documents ont ensuite été remis à jour au fur et à mesure du déroulement du projet.

La structuration initiale a servi de support durant toute la durée du projet, l'organisation nous a permis de gagner en temps et en efficacité. Au début du projet, nous avons défini les *Good Practice* à respecter pour assurer la robustesse du projet. Le nommage des classes, des méthodes, des attributs, l'organisation générale des fichiers .cpp et .h, tout avait été explicité afin de rendre le plus intuitif possible le développement.

1.2 Compréhension du sujet

1.2.1 Type de *Note*

Une classe Note est définie dans l'UML, dont héritent trois différents types de notes : les articles, les tâches et les enregistrement. Chacune des sous-classes de Note possède ses propres attributs tels que la priorité et la date limite pour la tâche et le texte pour l'article. Deux énumérations définissent respectivement le statut d'une tâche et le type d'un enregistrement.

La classe Note possède des attributs communs à l'ensemble de ses sous-classes : un identifiant unique par note, identique pour toutes les versions d'une même note, un titre et une date de création et de modification. Deux notes différentes ne peuvent posséder le même identifiant, l'unicité de l'identifiant assure la cohérence des données. Deux notes possédant le même identifiant sont nécessairement deux versions d'une même note.

Une note possède également deux attributs booléens spécifiant si une note est supprimée ou non, archivée ou non et un tableau de références sur d'autres notes. Il y a une relation d'agrégation entre la classe Note et elle-même car une note peut contenir un ensemble de notes en référence. Le tableau de référence est une QList contenant les ID des notes référencées. Il a été choisi de retenir l'ID et non une référence sur la note afin de faciliter la gestion des différentes versions d'une note dans le tableau des références.

1.2.2 Versions de notes

Pour réussir à gérer toutes les versions d'une note il nous fallait une structure spécifique. Nous avons donc choisi de nous tourner vers la structure de liste prédéfinie dans la STL. Une même note est ainsi regroupée à un seul et même endroit, peu importe le nombre de versions qu'elle contient.

L'avantage principal d'avoir choisi une liste est que cette structure est pré-implémentée, notamment en utilisant des algorithmes assurant une bonne performance. Cette structure nous permet également d'insérer nos nouvelles versions au début de la liste. Elles sont donc organisées de la plus récente à la plus ancienne dans la liste.

La liste des versions nous permet également d'avoir à disposition un itérateur pour pouvoir la parcourir simplement.

1.2.3 *Manager* de notes

Pour pouvoir gérer l'ensemble des notes nous avons implémenté une classe NotesManager qui, comme tout manager est sous forme d'instance unique. Nous avons également implémenté un itérateur personnalisé ainsi il nous est possible par la méthode *current()* de récupérer la dernière version d'une note que nous pouvons considérer comme version active, et par la méthode *liste()* de récupérer la QList complète contenant toutes les versions.

La classe NotesManager est composée par la classe Note : il gère la durée de vie des notes, si le manager est supprimé, l'ensemble des notes qu'il gère l'est aussi.

Les notes sont stockées dans le manager par un double pointeur de QList de pointeur de Note (*QListe<Note*>***). Le double pointeur permet d'avoir un tableau dynamique de pointeurs de QList. Les pointeurs de QList et de Note sont essentiels car ils permettent d'agir directement sur ce qui nous intéresse au lieu de manipuler des copies.

Il faut préciser que les notes ne sont pas forcément consécutives dans le manager, en effet si au début c'est le cas, lorsque l'on vide la corbeille la suppression des notes va engendrer des listes vides dans le manager. C'est pourquoi lors de l'ajout d'une nouvelle note celle-ci se retrouve à la fin du tableau sauf si il existe une liste vide auquel cas cette nouvelle note prendra donc la place d'une ancienne note supprimée.

1.2.4 *Relations* entre notes

La classe NotesCouple est en agrégation avec la classe Relation : une relation contient un ensemble de couple de notes. Dans le cas où l'ensemble des couples d'une relation est supprimé, la relation existe toujours mais est vide. Elle pourra être repeuplée par de nouveaux couples.

La classe NotesCouple possède deux attributs noteX et noteY pointant chacun sur une note du couple, un attribut pour le label du couple et un attribut booléen spécifiant si le couple est symétrique. C'est à dire que la relation va de noteX vers noteY et réciproquement. La classe NotesCouple est composée par la classe Notes : si les notes sont supprimées, les couplent n'existent plus non plus.

La classe Relation possède un titre, une description et un ensemble de couple de notes. Dans l'implémentation de cette classe, un Design Pattern Iterator est utilisé pour faciliter la manipulation des données.

1.2.5 *Manager* de relation

Comme pour gérer l'ensemble des notes, une classe manager est également utilisée pour gérer l'ensemble des relations. Un Design Pattern Singleton et Iterator sont utilisés pour assurer la cohérence des informations et la facilité d'utilisation du manager.

La classe RelationManager est composée par la classe Relation : si le manager est supprimé, l'ensemble des relations qu'il gère l'est aussi.

1.3 UML

Une première version de l'UML a été proposée durant la phase de pré-développement (tout début de la conception). Cette version a été largement modifiée et complétée afin de donner la dernière version de l'UML donnée ci-après.

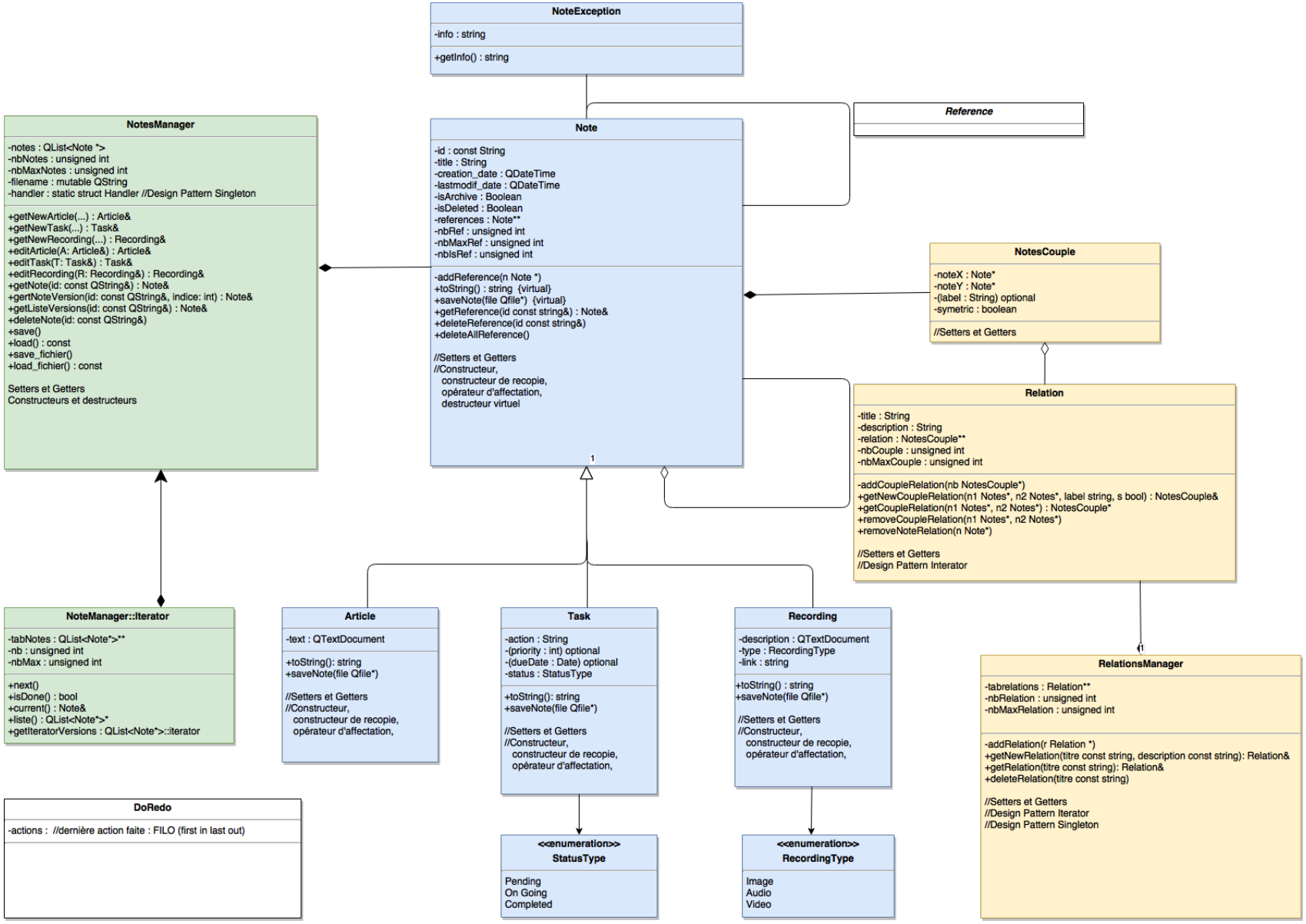

FIGURE 1 – UML version finale

2 Phase d'Implémentation

2.1 Interface

L'application Plurinote démarre sur une fenêtre principale divisée en 3 fenêtres distinctes : une dédiée à la visualisation globale des notes (partie gauche), une pour la vue détaillée des notes (partie centrale) et une pour l'affichage des relations d'une note (partie droite).

Un menu situé en haut de l'application permet d'ouvrir un fichier de note, de créer une nouvelle note, de supprimer, archiver ou fermer une note, de visualiser ou éditer les relations entre les notes et de sauvegarder les notes dans un fichier.

Un menu déroulant dans la barre des menus permet également d'effectuer une partie de ses actions, mais également de rétracter la vue des relations et de vider la corbeille.

2.1.1 Visualisation globale des notes

Le volet de gauche est lui-même divisé en trois blocs :

- Le premier permet de visualiser l'ensemble des notes créées et leurs versions sous forme de liste.
- Le second permet, également sous forme de liste, la visualisation des notes archivées. Il est possible de supprimer une note directement depuis cette fenêtre.
- Le troisième, identique au second, permet quant à lui la visualisation des notes supprimées. Il est possible d'archiver une note directement depuis cette fenêtre.

Un double clique sur une note de ces trois listes amène à sa visualisation détaillée dans la fenêtre centrale.

2.1.2 Vue détaillée de note

La partie centrale de l'application donne une vue détaillée d'une note. L'ensemble des informations la concernant y sont visible : ID, titre, texte, priorité, date limite, mais également les références que cette note a avec d'autres notes. Les informations concernant le type de note, sa date de création et de dernière modification sont inscrites en bas de la page.

Une note est par défaut accessible en lecture seule, mais si son édition est possible (c'est à dire qu'elle n'est ni archivée ni supprimée) alors un bouton en haut de la page propose son édition. Dans le cas contraire, un bouton propose sa restauration, qui entraîne la création d'une nouvelle version. Chaque fois qu'une note est éditée, une nouvelle version est ajoutée à la liste de ses versions, visibles dans la partie gauche de l'application. En cours d'édition, changer de note ou effectuer une autre action entraînera automatiquement la sauvegarde des modifications apportées.

Les notes de type enregistrement possèdent des méthodes spécifiques permettant de lire au sein de la vue centrale une vidéo, une image ou un audio ajouté par l'utilisateur.

2.1.3 Affichage des relations

Les relations qui peuvent lier une note à d'autres sont visibles dans la partie droite de l'application. Lorsqu'une note est affichée en vue détaillée, l'ensemble des notes qui lui sont liées en relation descendante (ou ascendante) sont visibles dans l'onglet haut (ou bas) de la partie relation. La liste de note en relation est active, il suffit de double-cliquer sur une des notes pour qu'elle s'affiche dans la partie centrale.

Double-cliquer sur le nom de la relation par laquelle ces deux notes sont reliées permet de la modifier : le nom, la description, l'ajout, la suppression d'un couple sont possibles.

2.1.4 Création, gestion et suppression d'une Note

La création d'une nouvelle note se fait par sélection du bouton "Nouvelle Note" situé dans le menu en haut de l'application. Une fenêtre s'affiche alors permettant d'entrer les informations relatives à la note et de sélectionner le type de note que l'on souhaite créer.

Une nouvelle version d'une note est créée après chaque édition de celle-ci. La liste des versions des notes est visible dans le dock haut gauche de l'application. Quitter une note en cours d'édition entraînera automatiquement sa sauvegarde sous la forme modifiée.

La suppression d'une note se fait de la même manière que sa création, en sélectionnant le bouton correspondant en haut de l'application. Plusieurs notes peuvent être supprimées en même temps. Si la note en question n'est pas référencée par d'autres notes, elle est déplacée dans la corbeille. Sinon elle sera automatiquement archivée. Lorsque toutes les notes référençant la note en question sont supprimées, il est enfin possible de la supprimer.

La suppression d'une note entraîne son retrait des relations qui la liaient. Les couples dont elle faisait partie n'existent plus non plus par conséquent.

Les notes de la corbeille ne sont pas modifiables mais sont restaurables et archivables. L'ensemble de ses notes peut être définitivement supprimé lorsque l'utilisateur désire "vider la corbeille". Cette action lui sera également automatiquement proposée à l'arrêt de l'application.

2.1.5 Création, gestion et suppression de Relation

L'édition de nouvelles relations pour une note se fait en sélectionnant le bouton correspondant dans le menu. Le titre, la description et les relations en relation avec la note sélectionnée sont à renseigner. Si une note est choisie à la fois comme en relation ascendante et ascendante, le couple sera automatiquement défini comme symétrique. Après création de la relation et des couples associés, il est proposé d'entrer un label propre à chaque couple en relation.

2.1.6 Création, gestion et suppression de Référence

Une référence entre une note et une autre est automatiquement repérée lorsque l'utilisateur entre "\ref{id}" dans un champ texte d'une note, que ce soit lors de sa création ou de son édition. Si l'id spécifié est reconnu, la références est directement créée. Les références des notes sont visibles au bas de leur vue détaillée.

Pour supprimer une référence, il suffit de retirer du champ de texte le "\ref{id}".

2.2 Sauvegarde du contenu de l'application

La sauvegarde du contenu de l'application, c'est-à-dire des relations et des références, se fait grâce aux fichiers XML. L'avantage principal d'un fichier XML est qu'il est composé de balises ce qui nous permet de le parser facilement dans le cas où l'on souhaite charger un fichier.

2.2.1 Sauvegarde des notes et relations

Pour sauvegarder les notes et les relations dans un fichier XML nous avons utilisé plusieurs balises comme *NoteVersions* qui permet de distinguer une seule note avec toutes ses versions, *article*, *task* et *recording* qui permettent d'identifier une note ou encore *relation* qui permet d'identifier une relation. Pour plus de détails, un exemple "type" est disponible en annexe.
La sauvegarde des notes est automatiquement faite lors de la sortie de l'application.

2.2.2 Chargement d'un fichier

Pour charger un fichier XML, il nous suffit de le parser en identifiant chaque élément grâce aux balises qu'il contient. Suivant les éléments que l'on a lu, il nous suffit de personnaliser le traitement effectué afin de remplir nos deux managers

2.2.3 Sauvegarde du contexte

À chaque ouverture de l'application, celle-ci va recharger le dernier jeu de données utilisé.

3 Maintenabilité et adaptabilité du code

Ayant respecté tout au long de développement un ensemble de *Good Practice* définies à l'avance, le code de notre projet est facilement maintenable. Les fonctions, méthodes et attributs sont nommés selon des critères précis, selon les actions qu'ils effectuent, les objets sur lesquels ils agissent, etc. La documentation précise et la cohérence de nos classes facilitent l'ajout de nouvelles fonctions et fonctionnalités.

3.0.1 Nouveau type de note

Dans le cas d'un développement plus important de l'application, la création de nouveaux types de notes, par exemple, est facilement envisageable et implémentable. Il suffirait de créer de nouvelles classes découlant de Note : le manager pourrait les gérer et la quasi totalité des fonctions seraient déjà fonctionnelles pour ce nouveau type de note : il n'y aurait qu'à ajouter une méthode qui créerait au sein du manager une note du nouveau type. Cette méthode est similaire à celles déjà créées pour les articles, tâches et notes, avec la différence des arguments spécifiques aux attributs du nouveau type.

3.0.2 Diversifier les relations

Les relations entre les notes pourraient être diversifiées afin de devenir des relations entre des objets. Bien qu'elles soient actuellement constituées d'un tableau de couples de notes, il serait possible de modifier la classe NotesCouple pour créer une classe abstraite Couple qui référerait deux objets ObjetX et ObjetY, peu importe leur type. La classe Relation pourrait alors être adaptée en *Class Template* : elle ne requière pas de Type, peut s'adapter à n'importe quelle classe fournie à conditions de bien définir certaines méthodes.

3.0.3 Agrandir l'utilisation des regex

Les références, telles qu'implémentées actuellement, consistent en une liste de QString. La méthode *generateRef* recherche dans chaque champ de texte une expression régulière et identifie l'ID qui y est inscrite avec la liste des ID des notes actives. En créant une nouvelle base de données d'auteurs, de lieux ou de thèmes, de nouvelles regex pourraient être proposées afin de reconnaître automatiquement ces informations dans une note. L'utilisateur aurait alors la possibilité d'entrer \ref{*id*}, mais également \location{*id*}, \author{*id*}, \topic{*id*}, afin de trier ensuite les notes par auteurs, lieux, sujet ou autres.

3.0.4 Proposer une utilisation en réseau

Pour une utilisation en réseau, il serait possible de migrer le système de sauvegarde et chargement à partir de fichier afin de le faire passer sur une base de données sur serveur en ligne. Le format XML des fichiers de note est tout à fait adapté à ce type d'utilisation, les fonctions *save* et *load* sont en revanche à ré-écrire (mais uniquement celles-ci, le reste étant indépendant de la méthode choisie).

3.0.5 Modularité de l'interface graphique

Un affichage multi-onglet pourrait être intéressant lors du parcours de deux notes en relation. L'interface est implémentée de telle manière à ce que chacune de ses parties est indépendante : il serait très facile de demander plusieurs instances d'un même type de Dock sur une même vue. L'interface de l'application est très modulable et permet une multitude de visuel : modification de la place des fenêtres, du nombre de fenêtres, du nombre d'instance d'une même fenêtre, etc.

Par ailleurs, le choix qui a été fait d'implémenter l'interface indépendamment des fonctions sur les notes nous permet d'assurer la maintenabilité de l'application. Il est aisé de modifier la structure des notes, par exemple, ou d'ajouter de nouvelles structures, sans impacter l'interface graphique.

4 Conclusion

Ce projet a été l'occasion pour les membres du groupe de se confronter au développement complet d'une application, de sa conception à son implémentation, jusqu'à sa documentation. Les méthodes informatiques abordées en cours et en TD, comme les Design Pattern, ont pu être approfondies et mieux comprises. Si la nécessité d'un Design Pattern Singleton ou Itérator semblait peu clair au début, ils ont rapidement étaient assimilés et sont devenus nécessaires au bon fonctionnement de notre projet.

Bien que nous ayons eu l'impression d'être organisés tout au long du semestre, la semaine précédant le rendu nous a semblé assez intense. Nous avons réalisé l'importance de commenter le code de façon lisible et explicite, à la fois pour l'usage des autres membres du groupe mais également pour son propre usage, car il peut arriver de retrouver une fonction codée par soi-même et dont on ne comprend plus l'utilité.

Le travail de groupe au sein de ce projet a également été très formateur : après la phase de conception, nous nous étions répartis l'implémentation de manière assez arbitraire. Très vite, nous avons réalisé que nous étions devenus très dépendants les uns des autres. L'un s'occupait des notes, un autre des managers, un dernier de l'interface : chacun "passait commande" des fonctions dont il avait besoin à la personne informellement responsable de cette partie.

Un tel projet informatique est à la fois essentiel pour bien comprendre et assimiler les notions théoriques vues au sein de l'UV, mais est également très formateur concernant l'apprentissage de la rigueur en informatique et la gestion de son travail au sein d'un groupe.

5 Annexe

5.1 Différents visuels de l'interface Plurinote

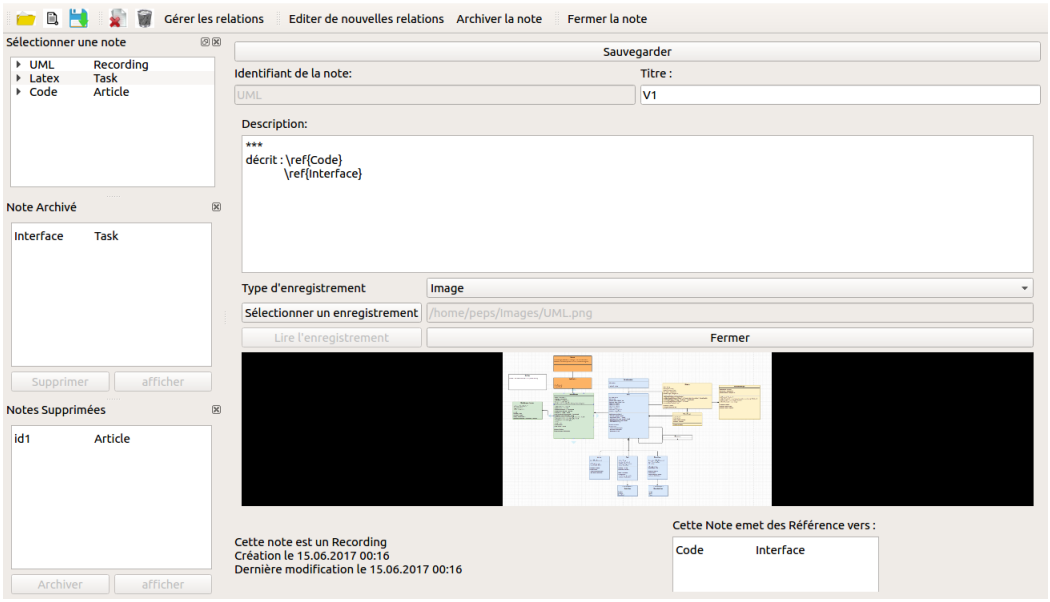

FIGURE 2 – Une vue détaillée d'une note enregistrement avec référence dans Plurinote

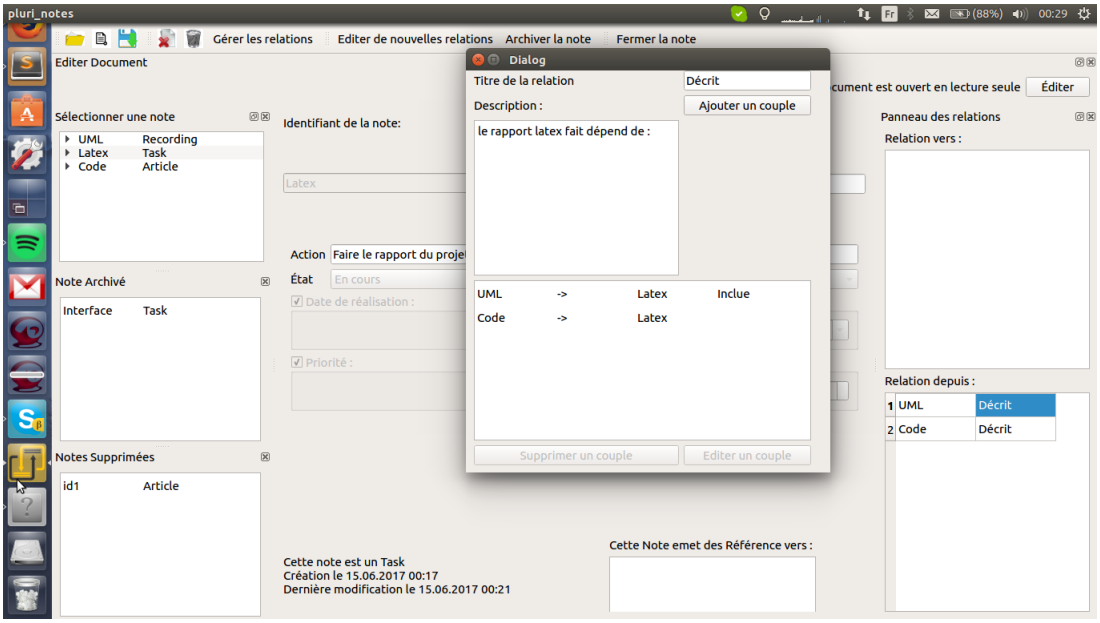

FIGURE 3 – Édition de relations dans Plurinote

5.2 XML "type"

```
<?xmlversion="1.0" encoding="UTF-8"?>
<NoteManager>
    <NoteVersions>
        <article>
            <id> </id>
            <title> </title>
            <c_date> </c_date>
            <lm_date> </lm_date>
            <isArchived> </isArchived>
            <isDeleted> </isDeleted>
            <text> </text>
        </article>

        ...
    </NoteVersions>
    <NoteVersions>
        <task>
            <id> </id>
            <title> </title>
            <c_date> </c_date>
            <lm_date> </lm_date>
            <isDeleted> </isDeleted>
            <action> </action>
            <priority> </priority>
            <d_date> </d_date>
            <status> </status>
        </task>

        ...
    </NoteVersions>
    <NoteVersions>
        <recording>
            <id> </id>
            <title> </title>
            <c_date> </c_date>
            <lm_date> </lm_date>
            <isArchived> </isArchived>
            <isDeleted> </isDeleted>
            <description> </description>
            <type> </type>
            <Link> </Link>
        </recording>

        ...
    </NoteVersions>
    <relation>
        <title> </title>
        <description> </description>
        <notecouple>
            <couple>
                <notex> </notex>
                <notey> </notey>
                <label> </label>
```

```
                <symetric> </symetric>
            </couple>

            ...
        </notecouple>
    </relation>

    ...
</NoteManager>
```