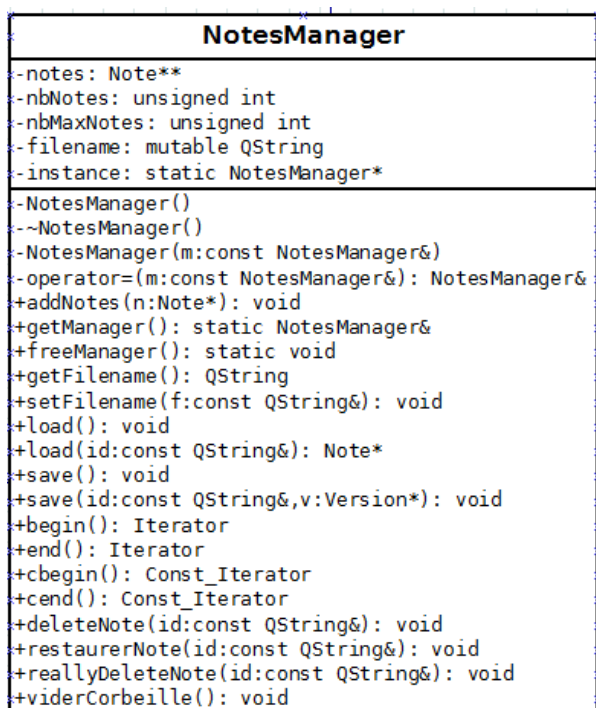
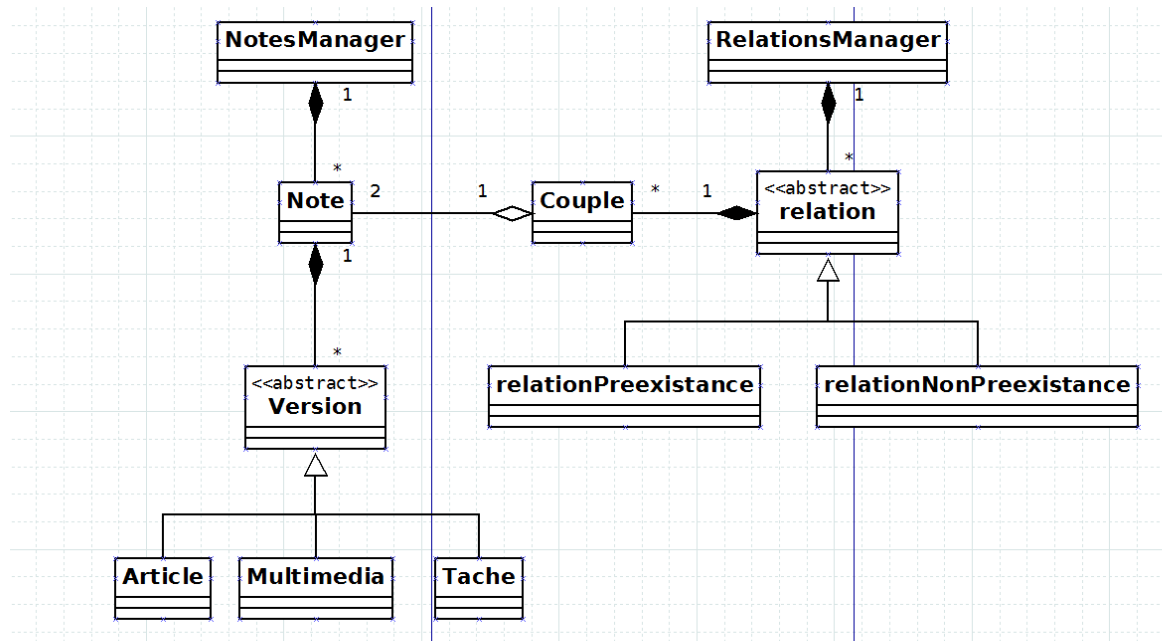


LO21

Rapport de projet : PluriNotes

Afin de développer notre application PluriNotes, nous avons mis en place l'architecture suivante :



Les notes sont gérées par le *Singleton* **NotesManager** qui se charge d'ajouter et de supprimer les instances de la classe **Note**. NotesManager est donc composée de **Note** et se charge de leur cycle de vie. Nous avons rajouté un *Iterator* afin de parcourir les notes créées.

La classe **NotesManager** joue également le rôle de corbeille et permet ainsi de supprimer définitivement toutes les notes placées dans la corbeille à l'aide de la méthode `viderCorbeille` ou d'en supprimer une seule à l'aide de `reallyDeleteNote`.

`DeleteNote` permet de mettre une note dans la corbeille et `restaurerNote` permet de restaurer une note.

Les méthodes `load()` et `save()` permettent de parcourir les notes et de charger/sauvegarder leurs informations dans un fichier.

La méthode `save(id,v)` permet d'ajouter une version à une note.

La méthode `load(id)` permet de renvoyer une référence d'une note.

Note
-creationDate: QDateTime -id: QString -etat: EtatNote -versions: Version** -nbVer: unsigned int -nbMaxVer: unsigned int
+Note(id:const QString&) +Note(dC:QDateTime,id:const QString,etat:EtatNote) +~Note() +getId(): const QString& +getDateCreation(): const QDateTime& +getEtat(): const EtatNote& +setEtat(status:const EtatNote): void +addVersion(v:Version*): void +setVersionActive(v:Version*): void +VersionActive(): Version* +getNewNote(id:const QString&,v:Version*): static Note* +getVer(titre:const QString&): Version* +getVerParDate(date:const QString&): Version* +save(stream:QXmlStreamWriter): void +begin(): Iterator +end(): Iterator +cbegin(): Const_Iterator +cend(): Const_Iterator

La classe **Note** permet de gérer les différentes versions d'une note. Elle a la responsabilité de leur cycle de vie : on a donc une composition entre **Note** et **Version**. Un *Iterator* permet de parcourir les différentes versions d'une note.

La méthode `save(stream)` permet de parcourir les différentes versions et de sauvegarder ses informations dans un fichier.

La classe **Version** permet de gérer les différentes versions des notes. La classe est abstraite afin de différencier trois types de notes : les articles, les tâches et les images.

Nous avons donc créé un héritage pour ces trois classes (**Article**, **Multimedia**, **Tache**) puisqu'ils partagent beaucoup d'attributs/méthodes tels que le titre ou la date de modification. Une note étant responsable du cycle de vie de ses versions, nous avons mis en place une composition entre **Note** et **Version**. Nous avons rajouté un *Iterator* afin de parcourir les versions d'une note.

La méthode `save` est virtuelle pure puisqu'elle dépend du type de version utilisé.

<<Abstract>> Version
#titre: QString #dateModif: QDateTime
+Version(titre:const QString&="") +Version(titre:const QString,dM:const QDateTime) +getDateModif(): const QDateTime& +getTitre(): QString +setTitre(t:const QString&): void +<<virtual pure>> save(stream:QXmlStreamWriter&): void +<<virtual>> ~Version()

Article
-text: QString
+Article(ti:const QString,d:const QDateTime,te:const QString) +Article(ti:const QString="",te:const QString="") +getText(): const QString& +setText(t:const QString&): void +save(stream:QXmlStreamWriter&): void

Multimedia
-description: QString -nomFichier: QString -typeEnregistrement: Media
+Multimedia(ti:const QString,d:const QDateTime,f:const QString&,desc:const QString="") +Multimedia(f:const QString&,desc:const QString="") +getDesc(): const QString& +getNomFichier(): const QString& +getType(): const Media& +setDesc(d:const QString&): void +setType(type:const Media): void +setPath(f:const QString&): void +save(str:QXmlStreamWriter&): void

Tache
-action: QString -priorite: unsigned int -statut: EtatTache -dateEcheance: QDateTime
+Tache(dE:const QDateTime&,action:const QString&="",p:const int=0,e:const EtatTache=EN_COURS) +Tache(ti:const QString="",d:const QDateTime=currentDateTime(),dE:QDateTime=currentDateTime().addDays(1),action:const QString&="",p:const int=0,e:const EtatTache=EN_COURS) +getAction(): const QString& +getPriorite(): const unsigned int +getDateEcheance(): const QDateTime& +getStatut(): const EtatTache& +save(str:QXmlStreamWriter&): void

Afin de créer des relations entre 2 notes on crée la classe **couple** qui permet tout simplement de créer des couples de notes. Les couples sont composés de 2 notes mais une suppression d'un couple n'implique pas une suppression des notes. Nous créons donc une agrégation entre Couple et Note. Nous avons ajouté un *Iterator* afin de parcourir les références créées. Un couple est également composé de plusieurs références. On a donc une composition entre Couple et relation.

couple
-fromNote: Note* -toNote: Note* +label: QString +couple(fn:Note*,tn:Note*,lab:const QString&) +getFromNote(): Note* +getToNote(): Note*

RelationsManager
-relations: relation** -nbRelations: unsigned int -nbMaxRelations: unsigned int -filename: QString -instance: static RelationsManager* +addRelations(r:relation*): void -RelationsManager() ~RelationsManager() +RelationsManager(m:const RelationsManager&): RelationsManager& -operator=(m:const RelationsManager&): RelationsManager& +getRelation(titre:const QString&): relation* +getNewRelation(titre:const QString&,desc:const QString&,ori:bool=true): relation* +deleteRelation(titre:const QString&): void +getInstance(): static RelationsManager& +freeInstance(): static void +setFileName(f:const QString&): void +getFileName(): const QString& +load(): int +save(): void +begin(): Iterator +end(): Iterator +cbegin(): Const_Iterator +cend(): Const_Iterator

Les relations sont gérées par le *Singleton* **RelationsManager** qui se charge de créer et supprimer les instances de la classe **relation**. On a donc une composition entre RelationsManager et relation.

Un *Iterator* permet de parcourir les instances de la classe relation.

Les méthodes load() et save() permettent de charger/sauvegarder toutes les informations des relations comme le titre, la description...

La classe **relation** regroupe les couples d'articles qui sont en relation. Un *Iterator* permet de parcourir les couples de notes en relation.

Nous distinguons les relationPreexistence et les relationNonPreexistence. Les méthodes setTitle, setDescription et setOriente sont donc virtuelles pures.

<<Abstract>> relation
#titre: QString #description: QString #oriente: bool #couples: couple** #nbCouples: unsigned int #nbMaxCouples: unsigned int #deleteCouple(idNote:const QString&): void +addCouple(fn:Note*,tn:Note*,lab:const QString&): void +relation(ti:QString,desc:QString="",ori:bool=true) +<<virtual>> ~relation() +getTitre(): const QString& +getDescription(): const QString& +getOriente(): bool +<<virtual pure>> setTitle(ti:const QString&): void +<<virtual pure>> setDescription(d:const QString&): void +<<virtual pure>> setOriente(ori:bool): void +begin(): Iterator +end(): Iterator +cbegin(): Const_Iterator +cend(): Const_Iterator

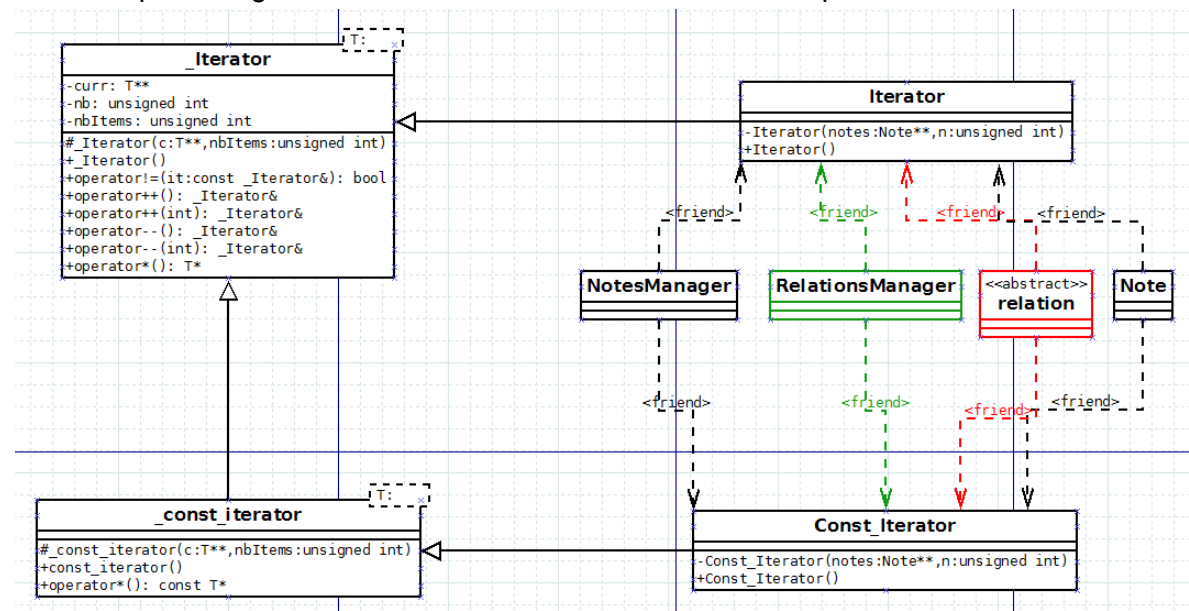
relationPreexistence
-instance: static relationPreexistence* -setTitle(ti:const QString&): void -setDescription(d:const QString&): void -setOriente(ori:bool): void -relationPreexistence() -relationPreexistence(re:const relationPreexistence&) -operator=(re:const relationPreexistence&): relationPreexistence& ~relationPreexistence() +getInstance(): static relationPreexistence& +freeInstance(): static void

relationNonPreexistence
+setTitle(ti:const QString&): void +setDescription(d:const QString&): void +setOriente(ori:bool): void +relationNonPreexistence(ti:const QString&,desc:const QString&,ori:bool=true)

Le fait d'utiliser beaucoup d'*Iterator* permet à notre application de gérer facilement les évolutions.

Ainsi, les classes **NotesManager**, **Note**, **RelationsManager** et **relation** en possèdent.

En effet, grâce aux itérateurs, on peut facilement ajouter ou supprimer des notes, versions...
Il faut simplement garder le nombre de notes et un tableau de pointeurs vers les notes.



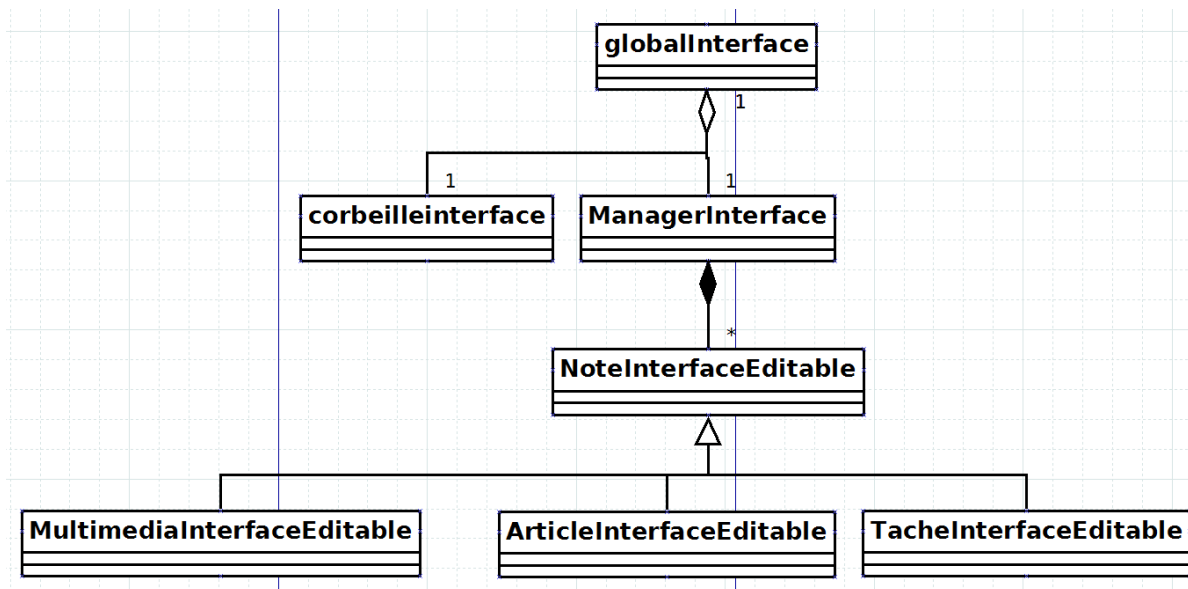
Un autre point qui rend notre application peu sensible aux changements est l'utilisation de la classe `relationPreexistence`.

Interface

A chaque classe fille de **Version**, on associe une interface graphique. Cette interface graphique hérite de la classe **NoteInterfaceEditable**.

A partir de la classe **Version**, on ajoute une méthode `créerInterface` qui renvoie un pointeur vers un **NoteInterfaceEditable**. Cette méthode sera définie dans les classes filles de **Version**. Ainsi, chaque version pourra appeler l'interface correspondante à son type (**Article**, **Multimedia** ou **Tache**). On peut donc appeler une interface sans connaître le type de la version.

L'interface a été faite de telle sorte que chaque partie graphique gère uniquement son contenu et le **GlobalInterface** gère les communications entre les différentes interfaces. Ainsi, l'application peut facilement gérer des changements et d'autres types de notes. En effet, on peut ajouter une nouvelle classe fille à la classe **Version** et lui associer une nouvelle interface graphique qui hériterait de la classe **NoteInterfaceEditable** assurant la compatibilité avec le reste du programme.



Fonctions non terminées par manque de temps

- Affichage des notes archivées (les notes supprimées ont tout de même leur attribut état changé)
- Affichage de l'arborescence des notes
- Ajout d'une nouvelle relation (on peut uniquement ajouter une référence lorsque l'utilisateur écrit `\ref{id}` où id est l'id de la note à référencer)
- Fonctions annuler et rétablir