UV L021

Rapport LO21

PluriNote P17

Capucine Prudhomme – Camille Dresse – Lucie Wartelle

15/06/2017

Table des matières

Introduction

Dans le cadre de l'UV LO21, nous avons dû concevoir et développer une application PluriNotes dont le but est la gestion et l'édition de notes. Celles-ci peuvent être composées de texte ou d'image selon leur type. A chaque fois que l'on édite une note, une nouvelle version est créée et l'ancienne est sauvegardée. Des notes peuvent avoir des relations entre elles, quand deux notes sont reliées par une relation, elles forment un couple.

Le but du projet était de créer une interface permettant de gérer la création, l'édition mais aussi la suppression de notes, et de gérer les relations que celles-ci peuvent avoir entre elles.

Ce rapport présente en première partie l'architecture de notre projet avec des explications quant à nos choix d'implémentation de classes et de méthodes avec leurs justifications. L'UML est disponible en annexe, il permet de mieux se rendre compte visuellement de l'architecture implémentée.

Description de l'architecture

Nous avons tout d'abord commencé par concevoir l'architecture de l'application demandée. L'UML correspondant est disponible en annexe. Nous allons ici détailler plus précisément les choix de conception qui ont été fait.

La classe Note

Nous avons choisi d'implémenter une classe Note qui a pour attributs un identifiant, un titre, une date de création et deux attribut booléens. Le premier 'active' permet de déterminer si une note est archivée ou non (une note archivée a son attribut active à false) et le second 'supprime' permet de déterminer si une note est placée dans la corbeille ou non. Une note est composée de une ou plusieurs versions. Pour représenter cela nous avons décidé d'inclure un tableau de Versions. La version active sera celle placée tout à la fin du tableau versions.

Le constructeur, le constructeur de recopie, et l'opérateur d'affectation sont en privé afin que seul l'instance de NoteManager puisse gérer le cycle de vie de ses objets Note, et pour empêcher la duplication de celles-ci. La classe NoteManager est déclarée amie de la classe Note.

Nous avons décidé d'instaurer une relation de composition entre les classes Note et Version, car un objet Note est responsable du cycle de vie des objets Versions de son tableau versions.

Le constructeur de Note créé une première version et initialise le tableau versions.

La méthode Editer() de Note prend en paramètre une nouvelle version et l'insère dans le tableau. On fait appel à la méthode clone qui permet de mettre dans le tableau la bonne classe fille correspondante. Pour gérer les différentes versions d'une note on a supposé qu'une note ne pouvait pas changer de type au moment de l'édition (c'est à dire qu'une note d'un article ne peut pas devenir une tâche par exemple) . Lorsqu'une note est éditée, la version ainsi créée devient la dernière version, on la place à la fin du tableau de versions de Note, c'est la version active.
La méthode Restaurer prend en paramètre une référence sur une version. Elle cherche dans le tableau de versions la version passée en paramètre, la stocke, décale le tableau vers la gauche puis la place à la fin du tableau de versions : elle sera donc la version courante, active.
La méthode GetDerniereVersion() renvoie une référence sur la dernière version du tableau.
La méthode VerifRef vérifie si dans la chaine de caractère passée en paramètre il y a "\ref{", puis récupère l'id de la note qu'elle référence. On cherche ensuite la note dans NoteManager puis on ajoute le couple dans la relation référence.
La méthode successeurs/prédecesseurs mets dans un tableau d'adresses de Note toutes les notes n2 qui sont reliées à la note n dans des relations sous forme de couple (n,n2)/(n2,n).

La classe NoteManager

Nous avons choisi d'implémenter une classe NoteManager dont la fonction est la gestion et l'édition des notes. Ainsi, un ensemble de notes est associé à un unique NoteManager qui est la seule entité autorisée à supprimer, ajouter ou éditer une note. La classe NoteManager est une classe amie de la classe Note afin de pouvoir utiliser ses méthodes privées et d'être la seule entité à en être capable. La classe NoteManager est un singleton car nous ne voulons qu'un seul objet capable de gérer les notes. La méthode getInstance() renvoie l'unique instance de NoteManager (managN) ou la créé si celle ci n'existe pas encore. Nous avons une relation de composition entre la classe NoteManager et la classe Note, car NoteManager est responsable du cycle de vie de ses objets Note.

De plus, pour pouvoir naviguer entre les notes, nous avons utilisé le design pattern Iterator.

La méthode viderCorbeille() utilise l'iterator de la classe NoteManager afin de parcourir l'ensemble des notes, et vérifie pour chacune d'elle que l'attribut supprime est à true : si c'est le cas, on parcourt la relation Reference grace à son iterateur afin de voir si la note est référencée au moins une fois (nous avons décidé de ne regarder que note2 pour les couples de Reference, car nous interprétons n1 est référencée par n2 comme un couple(n1,n2) et non (n2,n1)). Si aucune référence vers la note n'existe, on la supprime définitivement grâce à un delete. De plus, on supprime tous les couples de toutes les relations dans laquelle la note supprimée apparaissait. (Nous n'avions pas fait ceci dans la méthode supprimerNote() puisque cette méthode sert à rendre une note archivée, ou à la mettre dans la corbeille.

La méthode supprimerNote(Note& n) prend en argument la note à supprimer. Nous parcourons l'ensemble des couples de la relation Référence jusqu'à trouver un couple

référençant n (c'est à dire où n est la note2 du couple). Si on en trouve 1, alors n est référencée et on la note donc non active, on l'archive (attribut active à false), sinon on met l'attribut 'supprime' à true (la note est alors considérée comme dans la corbeille).

La classe Version

La classe Version ne possède qu'un attribut date de type datetime (nous avons pour cela fait un #include <ctime>) qui permet de distinguer les différentes versions d'une même note. Cette classe est ensuite gérée par héritage pour avoir les différents types de notes possibles : tâche, multimedia, Article.

C'est une classe abstraite, elle ne peut être instanciée, ses méthodes clone(), typenote(), et afficher() sont virtuelles pures et son destructeur est virtual pour respecter le polymorphisme.

Une version étant créée par NoteManager, qui est un singleton, on ne peut pas avoir deux versions créées en même temps, l'attribut date est donc clé.

Nous avons utilisés le design pattern Factory Method pour la méthode clone().

Les classes Article, Multimédia et Tache

Les classes Article, Multimédia et Tache sont des héritages de la classe Version, qui est abstraite, elles définissent donc les méthodes clone(), afficher() et typenote() virtuelles pures dans Version. Elles permettent d'implémenter les différents types de note et d'enregistrer les modifications de chaque version.

Nous avons choisi d'implémenter la classe Multimédia avec un attribut enum(Média) qui permet de savoir si c'est une vidéo, un audio ou une image.

La classe tâche a aussi un attribut de type enum(Etat) pour savoir si elle est en cours, terminée ou en attente. D'après l'énoncé, cet attribut doit être initialisé à 'en attente', nous l'initialisons donc ainsi dans le constructeur de Tache.

Ces deux types d'énumération ont été définis dans Version.h

La méthode clone ne prend pas de paramètre et alloue un objet du type de la classe fille en utilisant le constructeur de recopie . Elle renvoie un pointeur sur ce nouvel objet.

La methode afficher() retourne l'ensemble des infos de la version sous forme de QString afin d'être utilisée pour l'affichage dans l'interface Qt.

La methode typenote() retourne quand à elle le type de la version ("tache", "article", "media") sous forme de QString, afin de comparer cette valeur dans les methodes où elle sera appelée pour déterminer le type de la version traiter et ainsi choisir l'algorithme à appliquer.

La classe Relation

La classe Relation est un objet représentant un ensemble de couples de notes regroupés selon la relation que ses notes entretiennent entre elles. Cette classe est composée de la classe Couple, car elle est responsable du cycle de vie des couples de son tableau. En effet, deux notes ne forment un couple que si elles sont reliées par une relation.

Si l'utilisateur précise que la classe est non orienté on ajoutera les deux couples (x,y) et (y,x) de notes. On prend donc soin dans la méthode ajouterCouple() de WindowRelation de faire deux fois appel à addCouple(n1, n2, l) (puis addCouple(n2,n1,l2) pour créer le couple miroir) si la relation est non orientée.
Les méthodes addcouple() et suppcouple() rajoutent un couple et suppriment un couple respectivement dans la relation.
Comme pour la classe Note avec NoteManager, nous avons décidé d'utiliser une classe RelationManager, amie de Relation, et singleton, dont l'unique instance est la seule à pouvoir gérer les relations et utiliser leurs méthodes de construction, affectation, destruction.

Nous avons implémenté le design pattern iterator dans la classe Relation afin de pouvoir parcourir séquentiellement tous les couples d'une Relation. Attention, nous avons décidé d'implémenter un iterateur const, car nous voulons un accès en lecture uniquement d'après l'énoncé, nous ne voulons donc pas permettre à l'utilisateur de modifier les couples.

Deux méthodes, setTitre() et setDesc(), permettent de modifier le titre ou la description, elles prennent en paramètre un QString.

La méthode suppCouple() permet de supprimer un couple dans une relation. Si cette relation est non orientée, on supprime aussi le couple 'miroir'. Une fois le(s) couple supprimé(s), elle vérifie également si la note2 de ce couple est archivée, si c'est le cas on cherche si elle est encore présente en tant que note2 dans un autre couple de la relation Référence (on cherche si la note2 est encore référencée au moins une fois) via l'utilisation de l'itérator de Référence. Si ce n'est pas le cas, la méthode propose à l'utilisateur de supprimer la note. Si la relation était non orientée, on procède de même avec la note1 du couple passé en paramètre, puisqu'on a aussi supprimé le couple (note2, note1).

La classe RelationManager

De la même manière que pour la classe Note, nous avons choisi d'implémenter une classe RelationManager pour gérer et éditer des relations. La classe RelationManager est une classe amie de la classe Relation afin de pouvoir utiliser ses méthodes privées et d'être la seule entité à en être capable. De plus, pour pouvoir naviguer entre les relations, nous avons utilisé le design pattern Iterator.
La classe RelationManager est un singleton, afin d'avoir une seule instance d'objet s'occupant de gérer les différentes relations.

La méthode getInstance() renvoie l'unique instance de RelationManager (managR) ou la créé si celle ci n'existe pas encore.
A la création de RelationManager une première relation est créée c'est la relation référence qui est orientée, elle ne peut pas être supprimée. On ne la range pas dans le tableau de pointeurs de Relation relations, mais dans l'attribut Relation* Reference.

La méthode verifNoteRef() permet de vérifier si une note se trouve dans un des couples des relations existantes. Pour cela, cette méthode utilise l'iterator de NoteManager pour parcourir toutes les relations existantes et le const iterator de chaque relation pour parcourir chacun de ses couples. Si la note est encore présente dans l'un des couples des relations, la méthode renvoie un booléen true, sinon false.

La classe Couple

La classe couple sert à représenter l'alliance de deux notes par une relation. les objets de la classe Couple sont tous chacun gérés par une et une seule Relation, par une association de type composition.
Un objet Couple a pour attribut un label, qui est unique et de type int, et par deux pointeurs de Note (note1 et note2). Le constructeur par recopie de Couple est en privé car on veut interdire la duplication d'un Couple.

La méthode setLabel permet de modifier le label d'un couple elle prend en paramètre un argument de type int

L'interface Qt

Nous avons développé deux nouvelles classes : MainWindow qui sert à créer la fenêtre principale permettant toute la gestion des Notes. La deuxième est WindowRelation, qui est une fenêtre appelable depuis un onglet dans MainWindow, et qui permet la gestion des relations. Ces classes héritent toutes les deux de QMainWindow.

Évolutions possibles de l'architecture

Concernant notre architecture, nous avons implémenté celles ci de manière à pouvoir ajouter de nouvelles fonctionnalités en remettant le moins en cause possible l'architecture existante. Pour cela, notre architecture respecte plusieurs principes clés :

- **La décomposabilité :**

Nous avons divisé notre architecture en de nombreuses classes, suffisamment indépendantes les unes des autres, pour pouvoir travailler séparément sur chacune d'entre elles

Grâce à l'héritage par tâche, multimédia et Article de Version, on pourrait ajouter des types de Notes facilement en faisant une nouvelle classe qui hérite aussi de Version. De plus, avec la méthode clone() (qui est une implémentation du **design pattern Factory Method**) il n'y pas besoin de modifier la classe Version ou Note, mais seulement de définir la méthode clone() dans la nouvelle classe. La méthode virtuelle pure *notetype*() permet de récupérer le type de Version, il est possible de la redéfinir pour chaque nouvelle classe héritant de Version, de même que pour la méthode virtuelle pure afficher()

Version est une classe à part entière, on peut modifier sa structure sans modifier celle de Note, tant que la classe Note contient un tableau de pointeurs d'objets Version.

Le plus souvent dans notre architecture, deux classes ou modules qui interagissent entre eux sont liés par une relation de composition, ainsi une des deux classes est responsable du cycle de vie de l'autre, on a donc peu de risque de problèmes, et les interactions sont faciles à modifier.

Pour ajouter des fonctionnalités telles que des actions particulières, il y a de fortes chances, grâce à la forte décomposition de notre architecture en de nombreuses classes, pour qu'il suffise d'ajouter des attributs et méthodes dans une seule voir au pire deux classes.

Nous avons aussi implémenté des types Énumération. Par exemple, grâce à l'énumération Media, il est très facile de modifier le code pour ajouter un nouveau type possible de Multimédia. Il suffit d'apporter des modifications dans l'énumération et dans une ou deux méthodes du code, ce qui se fait rapidement et facilement.

- **La compréhensibilité :**

Un utilisateur peut comprendre facilement comment est organisé notre architecture, grâce à l'utilisation de noms explicites, à l'implémentation par compositions, (etc.) ce qui rend le code clair et facilement compréhensible pour pouvoir ensuite être développé et complété par n'importe quel utilisateur.

- **La continuité :**

Grâce à notre forte décomposition, un changement mineur dans la spécification du problème ne devrait nécessiter que de modifier une ou deux méthodes, qui au pire affectent l'architecture de une ou deux classes

De plus, nous avons essayé de faire l'architecture la plus complète, pratique et logique possible : normalement il n'y a pas de raison de rajouter de relation entre les classes qui ne sont pas déjà reliées dans notre architecture actuelle, car celles ci n'ont alors pas vraiment de lien logique entre elles.

De surcroît, il est possible d'accéder à toutes les classes de l'architecture à partir des classes manager (RelationManager et NoteManager), grâce à l'utilisation de pointeurs ou de méthodes virtuelles. Nous pouvons donc raisonnablement imaginer qu'il en serait de même si une classe venait à être modifiée, ajoutée, ou même détruite. Cela rend de plus le code plus sécurisé car les classes Note et Relation ne peuvent être utilisées que par NoteManager et RelationManager.

Dans le code, de nombreuses méthodes sont en privé, ce qui est bien pour la sécurité, et pour respecter le principe d'encapsulation.

Enfin, nous avons implémenté le design pattern iterator dans presque chaque classe, afin de ne pas exposer le code et l'interface de notre architecture, toujours dans une optique de sécurité optimale.

Interface Qt :

Nous avons décidé d'implémenter les actions liées aux notes, et celles liées aux relations dans deux fenêtres presque indépendantes. De ce fait, on peut imaginer ajouter un autre type de classe qui aurait sa propre fenêtre pour ses fonctionnalités.

Conclusion

En conclusion, ce projet nous aura permit de mettre en application les notions vues en LO21 concernant la programmation orientée objet avec ses fonctionnalités telles que les design patterns ou les notions de maintenabilité.
Nous n'avons pas pu implémenter la totalité des demandes du sujet, et nous nous sommes parfois rendu compte trop tard que nous aurions pu procéder différemment et plus efficacement.
Par exemple, nous avons commencé toute une implémentation assez tôt dans le semestre, nous n'avions pas encore bien étudié la STL avec la classe Vector par exemple, que nous aurions pu utiliser à la place des tableaux afin d'obtenir un code plus simple et sécurisé dans nos méthodes pour les compositions et agrégations de RelationManager, Relation, Couple, NoteManager, Version et Note. De même pour notre interface Qt, celle ci n'est pas la plus belle et la plus propre, mais nous avons choisi de faire quelque chose de plus "moche" mais de plus complet, où nous perdions moins de temps à déboguer. Nous espérons continuer à apprendre à développer des interfaces de manière à s'améliorer.
De plus, suite à des difficultés rencontrées, nous avons codé le XML mais les méthodes load() et save() ne marchaient pas, nous n'avons donc pas mis de fonctionnalités liées aux fichiers XML dans l'interface Qt.

De plus, nous n'avons pas eu le temps de créer le raccourics Ctrl+Z / Ctrl+Y, nous voulions implémenter le design pattern memento mais cela aurait pris trop de temps. Il en est de même pour le tri des tâches par date et priorité dans la liste des tâches du dock gauche de notre interface Qt.

Enfin, nous souhaitons précisé que nous avons oublié d'utiliser des références /{idy} dans notre vidéo, même si nous avons bel et bien implémenté une méthode verifRef() qui ajoute la référence si il y a \{idy} dans un texte, et cette méthode fonctionne.

Annexes

Lien Github

https://github.com/camilledrs/LO21_PluriNotes

UML