LO21: Rapport du projet PluriNotes

Yannis Ben Ouaghrem - Romain Fayolle - Maxime Lucas

15 juin 2017

Table des matières

I Introduction

```cpp
int main(int argc, char *argv[]) {

    std::cout << "Hello World !"" << std::endl;
}
```

Préambule

Bienvenue dans notre rapport de projet concernant la réalisation d'une application permettant une gestion ergonomique de notes diverses : Articles, Multimédias, Tâches, ...
Ce document vous permettra de comprendre nos choix d'architecture, le déroulement du projet, ainsi que les problèmes rencontrés.
Au niveau des ressources, nous avons utilisé :
- Git, pour le travail collaboratif et le versioning
- QtCreator, comme IDE
- Draw.io, pour la réalisation des diagrammes UML
- TexMaker, pour la rédaction de ce document

Groupe de travail

Composé de trois étudiants de formations différentes, notre groupe de travail était assez hétérogéne en terme de compétences en C++, en début de projet :
- Yannis Ben Ouaghrem : GI02 - Tronc Commun
- Romain Fayolle : GI02 - DUT Informatique
- Maxime Lucas : GI02 - DUT Informatique

Dépôt Git

Il vous est possible de consulter notre dépôt Git à l'adresse suivante :
https://github.com/maxime-lucas/utc-lo21-plurinotes

II Architecture et Développement

II.1 Modélisation UML

NB: Dans l'UML, nous utilisons les types génériques : string, datetime, ... Ce n'est que lors de l'implémentation que nous avons choisi d'utiliser les types propres à Qt (QString, QDateTime,...)
Aussi, la modélisation UML étant déjà très lourde, nous avons préféré proposer un diagramme UML très allégé. Pour plus d'informations sur une classe, vous pouvez vous référer à la documentation Doxygen.

Package NOTES

Dès la lecture du sujet, nous avons décidé de modéliser une classe mère **Notes**, contenant tous les attributs communs à chaque type de note à savoir les **Articles**, les **Multimédias** et les **Tâches**.
Il n'y a aucune fantaisie dans la modélisation de ces classes mis à part le fait que nous utilisons des énumérations pour le *type de multimédia* et le *statut d'une tâche*.
Toutes les notes sont gérées (instanciées et détruites) par un unique **NoteManager**, se dérivant sous trois formes :
- **ActiveNoteManager** : NoteManager qui s'occupe de toutes les notes actives.
- **DeletedNoteManager** : NoteManager pour modéliser la corbeille de l'application ; c'est-à-dire toutes les notes prêtes à être supprimées mais en attente de confirmation.
- **ArchivedNoteManager**. NoteManager permettant de référencer toutes les notes ne pouvant pas être supprimées car elles sont référencées par une relation spéciale.

Le NoteManager n'est composé que d'un attribut *tab* qui est en fait un vecteur de pointeurs sur des notes.
Nous avons préféré utiliser les vecteurs de la bibliothèque standard, plutôt que de réimplémenter notre propre design pattern *Iterator*. De plus, les vecteurs ont des fonctions très utiles, que nous avons pu utiliser sans modération.
Aussi, nous n'avons pas choisi d'utiliser le design pattern *Singleton* pour modéliser notre NoteManager (ainsi que les autres managers de l'application) puisque les managers sont uniquement instanciés par l'application principale. Aucune autre classe n'a donc besoin d'y avoir accès directement, ou le cas échéant, par le biais d'accesseurs.
En ce qui concerne les multimédias, nous ne stockons que le nom du fichier ; tous les fichiers multimédia sont stockés au même endroit.

Package VERSIONS

Au début, nous nous étions penchés sur le design pattern *Memento*. Bien que très simple d'utilisation, notre choix de stockage des données nous a poussé à stocker les versions d'une note, directement dans celle concernée. Nous reviendrons dans la suite de ce rapport sur notre choix de stockage des données.
Une **Version** est composée d'un attribut *numVersion*, unique pour une note donnée, ainsi que d'un attribut *state* de type *Note**. Ce dernier s'occupe de garder en mémoire l'état d'une note lorsqu'on a choisi d'en sauvegarder une version.
Une version pourra être rétablie ou supprimée par la note elle-même. Cette opération sera réalisée par le *controller* de l'application (cf. Framework MVC p.4).

Package XMLManager

Ce package ne comporte qu'une seule classe : le **XMLManager**. Il permet de faire le lien entre l'application et un fichier XML permettant de stocker les données.
Nous avons choisi en début d'application d'utiliser les fichiers XML plutôt qu'une base de données pour stocker les données car nous pensions qu'une base de données serait bien trop gourmande pour les besoins de notre application. Grossière erreur.
En effet, même si Qt a ses propres classes pour le parsing des fichiers XML, il n'en est pas plus simple de créer une telle interface. Avec le XMLManager, il a fallu développer des fonctions de recherche, d'insertion, de suppression, ... alors que nous aurions très bien pu nous débrouiller avec des requêtes SQL qui nous auraient fait économiser de bonnes lignes de codes.
En revanche, malgré son côté robuste, le XMLManager demeure l'une des classes les plus modulaires de notre application. Avec un peu plus de temps, nous pourrions très bien remplacer le système de gestion des données par un SGBD très léger type sqLite, et il n'y aurait pas tant de changement à faire que cela.

Package RELATIONS

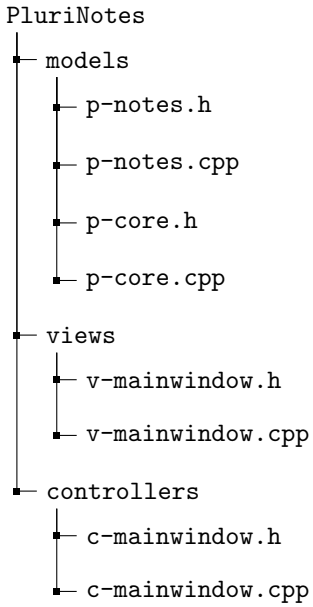
C'est dans ce package que se trouvent toutes les classes permettant de gérer les relations, au sens mathématique du terme, entre les différentes notes de l'application.
Un **Couple** est composé d'un *id* (unique), d'un *label* (modifiable) et de deux *notes* : noteX et noteY. Comme les couples peuvent appartenir à des relations orientées, il est utile d'orienter aussi le stockage des deux notes contenues dans un couple. En effet, si la relation est non-orientée, alors on ne se préoccupe pas de si la note est noteX ou noteY, alors que si la relation est orientée, l'ordre a un sens.

Ainsi, une **Relation** est composée d'un *id* (unique), d'un label (modifiable), d'un *ensemble de couples* (vecteur modifiable) ainsi que d'un attribut booléen *isOriented* et un attribut booléen *isReference* qui permettra de savoir si on instancie l'unique relation **Référence** de l'application.
Les relations sont gérées par un **RelationManager** qui s'occupera de les instancier, et aussi d'instancier la relation **Référence**.

Package CORE

Composé que d'une seule classe, c'est le point d'entrée de toute l'application. En effet, **PluriNotes** est la classe qui compose tous les managers : **NoteManager**, **XMLManager** , **RelationsManager**, ... et qui permet l'accès à toutes les données.

II.2 Frameworks : Qt et MVC



Nous avons choisi d'implémenter le framework MVC pour une meilleure structuration et un travail collaboratif plus aisé. De ce fait, pendant que certains s'occupaient du traitement des fonctions dans le *controller*, d'autres pouvaient s'occuper de l'interface graphique en développant les classes dans le dossier des *views*, ou encore gérer les structures de données, et le stockage à l'aide des *models*.
Nous n'avons pas choisi de représenter les *views* et le *controller* dans l'UML pour ne pas surcharger le rendu.
De plus, le développement des vues reste très propre au framework Qt et ne rend pas cette partie modulable ou adaptable à un autre framework UI.

III Modularité et Evolution de l'application

Atout du framework MVC

Le MVC permet de rendre l'application très évolutive. En effet, un tel framework permet de pouvoir améliorer l'interface graphique sans se soucis de l'impact sur les structures de données. La maintenabilité est donc garantie et est plus aisée.

Interface graphique améliorée

Nous avons choisi de fournir une interface graphique agréable et ergonomique. Vous avez la possibilité de choisir d'afficher ou non certains onglets, ou de passer d'une vue à l'autre simplement avec le clavier.
L'ajout des éléments tels que les articles, les multimédias, les tâches, les relations, ... peut se faire à l'aide de la barre d'outils, ou directement avec le menu supérieur de la fenêtre.
Sachez que la vue est elle aussi modulable ; par exemple, l'affichage des notes se fait à l'aide d'un widget central, héritant de la classe **V_CentralNote**. Nous étions amené à avoir un nouveau type de note, telle qu'une TODO-LIST, nous n'aurions qu'à créer une vue héritant de la classe **V_CentralNote** et nous pourrions très aisément adapter cette nouvelle vue dans l'application.

Affichage et stockage des multimédias

Les notes de type multimédia ont demandé un peu de réflexion. Pour enregistrer un nouveau multimédia, il suffit de choisir un fichier situé sur votre ordinateur à l'aide d'une fenêtre dialogue, et lors de l'enregistrement en XML, nous copions au préalable ce fichiers dans un dossier **Ressources** situé dans l'application. Ceci permet de ne pas perdre la référence vers un fichier si celui-ci venait à être supprimé ou déplacé. Bien entendu, dès lors que ce fichier est supprimé du dossier Ressources, nous perdons toute référence dessus.
Grâce à notre application, il est possible d'afficher un multimédia de type image, et d'afficher le contenu de cette dernière. En revanche, il ne nous est pas possible d'insérer un lecteur vidéo ou un lecteur audio pour les deux autres types de multimédia. Nous nous sommes renseignés sur la question et il semblerait que nous devions importer un module spécial de Qt, qui impliquait certaines directives de compilation et qui aurait rendu l'application beaucoup trop lourde pour très peu d'intéret.
Il n'est pas possible de modifier le fichier pointé par une note de type multimédia ; il faudra supprimer cette note et en recréer une nouvelle.

Sauvegarde de l'état de l'application

Pour cette problématique, nous nous sommes un peu écarté du sujet. En effet, nous n'avons pas choisi de sauvegarder l'état de l'application à sa fermeture, mais nous le sauvegardons à chaque manipulation sur les données. Ceci est aisément permis par le framework MVC.
En effet, le controller, dès lors qu'il doit modifier des données et dans les fichiers, et dans la vue, respectera toujours le même mode de fonctionnement :
- Appel de la fonction adéquate dans le XML Manager
- Modification dans les structures de données : Ajout dans un vector, Modification d'un champs, ...
- Rafraichissement des vues impactées par les modifications : On ne rafraichît pas la liste des Relations, si on effectue une modification sur une note.

Gestion des versions

Lorsque l'on modifie un champ d'une note, une version est automatiquement créée avec l'état précédent de la note, dès la confirmation de l'opération. Une note peut avoir un nombre indéterminé de versions. Grâce à l'interface, lors de la sélection d'une précédente version d'une note, il vous est demandé si vous souhaitez restaurer cette version ou la supprimer. Si vous décidez de supprimer cette version, toutes les versions qui auront été créées après celle-ci seront automatiquement supprimée.
Ces manipulations sont réalisées à l'aide du XMLManager (côté stockage) et avec la classe d'application principale (côté structure de données).

Axes d'amélioration

Au cours de l'évolution de notre projet, nous avons découvert des méthodes plus efficaces à mettre en place sur certaines parties. Par exemple le système de base de données aurait été plus simple avec SQLite qu'avec l'utilisation du XML.
La mise en place d'une session codage, par exemple une fois par semaine, avec les 3 membres du groupe peut aussi être une piste d'amélioration : le développement avance moins vite mais, chacun apportant son avis, il est plus juste.
Enfin, nous n'avons pas assez abusé de la méthode "papier, crayon, action" apprise en IUT. Cette méthode consiste

à suffisamment réfléchir sans coder afin que derrière aucun doute ne persiste lors de la phase de développement. En effet nous avons dû revoir plusieurs fois notre architecture pour inclure certaines fonctionnalités.