|  |  |
| --- | --- |
| Clément Fichmann – GI04  Alexandre Keil – GI02 | UTC  Printemps 2014 |

LO21 – Projet UTProfiler

Rapport

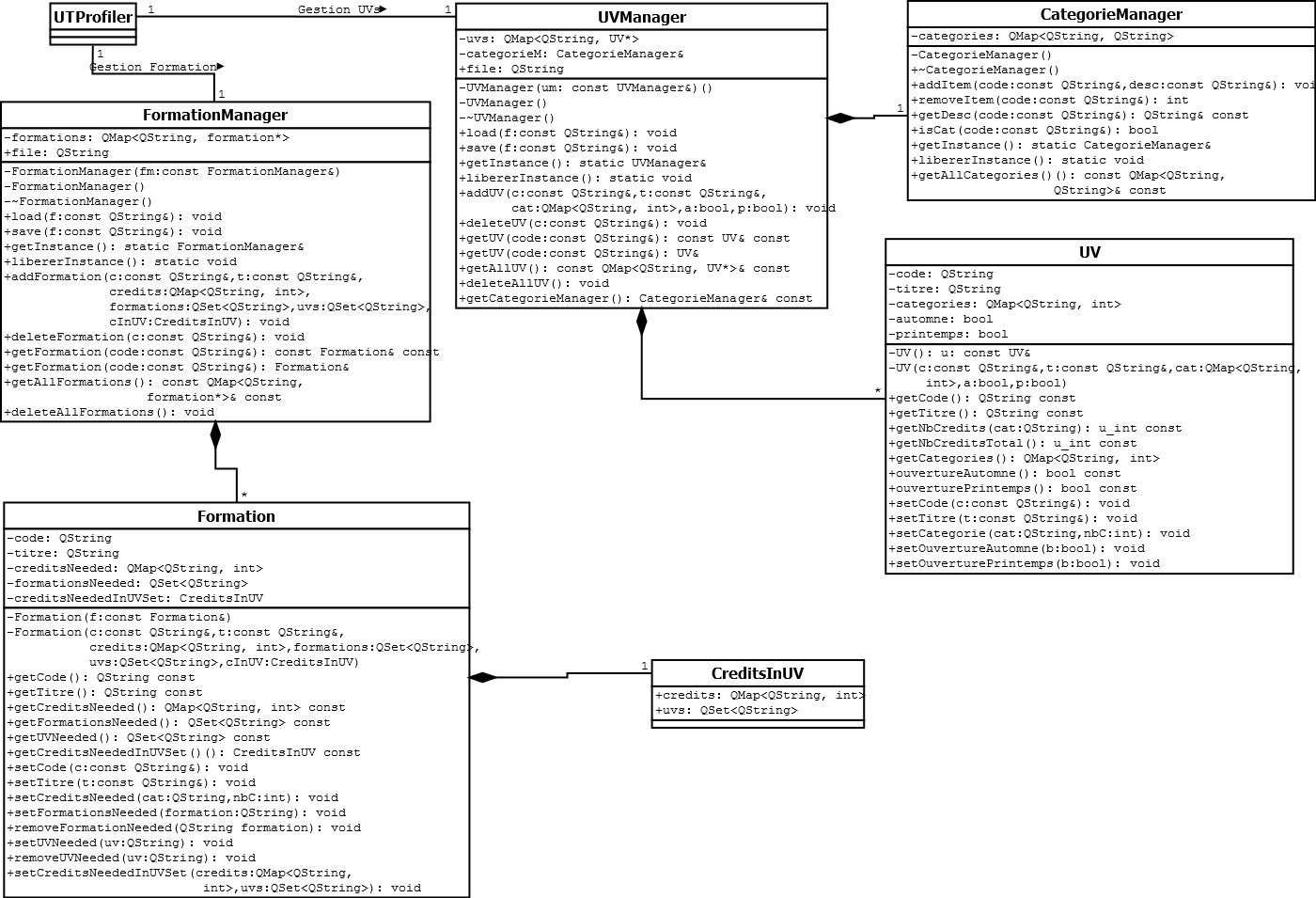
# Introduction

Dans le cadre de l’UV LO21 – Programmation Orientée Objet, il nous a été demandé de réaliser en binôme l’application UTProfiler, permettant de gérer le catalogue des UV de l’UTC, de créer, modifier et supprimer des cursus. L’application devait également offrir à un étudiant la possibilité de renseigner sa formation, ses UV passées et en cours de suivi, ainsi que d’être aidé pour ses choix futurs d’UV grâce à un algorithme de complétion lui suggérant des matières qui lui permettraient de compléter son profil commun de branche et son profil spécifique de filière.

Un des aspects nécessaires de l’architecture était qu’elle soit la plus adaptable possible. Nous avons par conséquent dû réfléchir aux solutions de conception permettant d’obtenir un programme malléable au changement.

Dans ce rapport, nous présenterons l’architecture que nous avons retenue pour notre version du projet, et les différents choix techniques que nous avons fait pour la conception et l’évolutivité du programme.

# Diagramme de classes



# Choix de l’architecture

## Gestion des UVs

L’un des aspects les plus délicats à gérer dans la conception de notre architecture a été celui de l’évolutivité des catégories de crédits. Le but était d’une part, de pouvoir ajouter facilement des catégories de crédits, et d’autre part de mélanger des crédits de différentes catégories pour une même UV.

Pour cela, nous avons défini la classe CategorieManager, à instance unique, qui ne possède qu’un seul attribut, nommé « catégories ». Cet attribut est une QMap<QString, QString>, fonctionnant comme un tableau dynamique qui, pour chaque clé contiendrait une valeur. Ici, la première QString, la clé, contient le code de la catégorie – par exemple « CS ». La 2e QString, la valeur, contiendra quant à elle le nom complet de la catégorie – par exemple « Connaissances Scientifiques ».

Lors du TD n°5 portant déjà sur la gestion des UV, les Catégories étaient prédéfinies dans une énumération, qui avait le défaut de n’être plus modifiable après sa définition, empêchant ainsi toute évolution des Catégories. A l’aide des méthodes adaptées, cette nouvelle classe nous permet désormais de créer, modifier et supprimer facilement des catégories. Par ailleurs, quand l’UVManager importe des UV à partir du fichier XML, le CategorieManager est automatiquement enrichi des catégories de crédits repérées dans le XML.

Nous nous sommes ensuite attaqué à la seconde partie du problème, qui concernait le mélange de crédits : pour reprendre l’exemple du TD5, la classe UV contenait à l’origine un attribut pour la catégorie et un autre pour le nombre de crédits, ne permettant donc d’attribuer à une UV différentes catégories de crédits.

Nous avons là encore utilisé une QMap nommée « Catégories », de structure <QString, int>. La clé de la QMap, de type QString, contient le code de la Catégorie, par exemple « CS » ; l’entier associé contient le nombre de crédits de la Catégorie clé pour cette UV.

En utilisant les formulaires adaptés, l’utilisateur peut combiner comme bon lui semble crédits de catégories existantes, et même créer de nouvelles catégories de crédits ; il est désormais possible d’accorder 4 crédits CS et 4 crédits TM pour une UV, par exemple.

## Gestion des formations

Les catégories de crédits d’UV étant désormais évolutives, nous sommes passés à la gestion des formations.

Nous avons créé une classe FormationManager, à instance unique (design pattern Singleton), contenant 2 attributs :

* « formations », une QMap<QString, formation\*>, privé ;
* File, un QString public.

Lors de l’ouverture du programme, la méthode load() de cette classe vient parser le XML pour récupérer la liste des formations, construit des Formations avec les informations lues dans le XML et remplit la QMap formations avec pour clé, le code (nom abrégé) de la Formation, et pour valeur un pointeur vers la Formation concernée.

Nous avons également créé la classe Formation, regroupant toutes les informations sur une Formation. Une Formation est caractérisée par :

* Un code (ou nom abrégé, par exemple « GI »)
* Un titre (ou nom complet, par exemple « Génie Informatique »)
* CreditsNeeded, une QMap contenant les crédits nécessaires à la validation de la Formation. Cette QMap possède des clés de type QString, contenant le code des catégories de crédits, et des valeurs entières, indiquant le nombre de crédits requis pour la catégorie en clé. Cette QMap reprend donc le principe de celle utilisée pour la gestion des crédits attribués par une UV.
* FormationNeeded, un QSet, fonctionnant comme une liste, regroupant les codes des Formations préalablement requises pour suivre la Formation courante.
* CreditsNeededInUVSet, une instance de la structure CreditsInUV que nous avons créé.

Cette structure CreditsInUV regroupe 2 attributs :

* Credits, une QMap fonctionnant sur le même principe que l’attribut CreditsNeeded de la classe Formation, en indiquant la quantité de crédits requis pour chaque catégorie de crédits ;
* Uvs, une QSet contenant un ensemble de codes d’UV.

CreditsNeededInUVSet regroupe ainsi les crédits de chaque catégorie devant être obtenus au travers d’un panel d’UVs afin de valider la formation.

On peut ainsi créer des Formations requérant d’avoir validé d’autres Formations, et nécessitant la validation de crédits de différentes catégories parmi une liste restreinte d’UV, comme par exemple pour le Profil Spécifique de Filière.

# Utilisation de l’application

Lors de l’ouverture du programme, une boîte de dialogue apparaît à l’écran, pour demander à l’utilisateur de renseigner les chemins des fichiers XML à analyser pour récupérer les données. Une fois ces données renseignées, en cliquant sur Valider, la boîte de dialogue s’efface pour laisser place à la fenêtre principale de l’application.

Il s’agit d’une fenêtre assez simple, utilisant la navigation par onglet pour passer d’une fonction à une autre. En haut d’un onglet, un bouton « Ajouter » permet de créer une UV ou une Formation, selon l’onglet. Le clic sur ce bouton provoque l’ouverture d’une nouvelle boîte de dialogue, contenant un formulaire identique à celui présent sur l’onglet de la fenêtre principale, permettant de renseigner toutes les informations nécessaires à la création ou à la modification de l’objet (UV ou Formation) sélectionné.

En bas des formulaires d’édition, sous le bouton de validation, on retrouve le bouton de suppression, demandant confirmation avant de supprimer l’objet sélectionné.

Enfin, la fenêtre principale contient un menu « Fichier » permettant :

* D’ouvrir des nouveaux fichiers XML d’UV ou de Formation (et donc de remplacer le contenu des objets UVManager et FormationManager par les éléments de ces nouveaux fichiers)
* De sauvegarder les modifications faites vers le fichier d’UV, le fichier des Formations, ou les deux.
* De quitter le programme. L’application demandera confirmation et proposera de sauvegarder les fichiers. A noter que le clic sur la croix de fermeture provoque les mêmes vérifications.

# Conclusion

Bien que nous n’ayons pas réussi à réaliser un programme remplissant toutes les fonctions demandées, ce projet a néanmoins été l’occasion d’appliquer à un cas concret et exigeant les principes enseignés en cours et en TD. Il nous a également permis de manipuler en profondeur le framework Qt, qui s’est avéré très puissant pour réaliser des interfaces graphiques à la fois simples et fonctionnelles.

Nous avons beaucoup travaillé sur la modularité et l’évolutivité des composantes des fonctions que nous avons réalisées – à savoir la gestion des UV et des formations –, ainsi que sur la conservation de la structure des fichiers XML lors de la sauvegarde des divers éléments manipulés.

Les fichiers XML, de par leur structure bien particulière, a nécessité des aménagements bien spécifiques de la structure du programme afin de pouvoir facilement rajouter de nouveaux types de données, telles que les catégories de crédits.

Il aurait peut-être été préférable de passer par un SGBD pour la sauveg arde des données, mais nous craignions de passer trop de temps à définir une base répondant au cahier des charges.

Ce projet demeure toutefois notre expérience la plus enrichissante en Programmation Objet, de par les exigences en termes de programmation, mais également d’adaptabilité du code.