Rapport - Projet LO21 : Logiciel de gestion de trésorerie pour une association

Client : Monsieur Thibaud DUHAUTBOUT

Co-réalisé par :

Garnier Kilian, GI02

Joubin Hippolyte, GI02

Dam Sébastien, GI02

Gervais Julien, GI02

Basta Ilan, GI02

Semestre P20 - 18/06/2020

Sommaire

Introduction

Dans le cadre de l'UV LO21, nous devions développer une application en C++, qui a pour but de gérer la comptabilité d'une association étudiante. Nous devions pour cela nous baser sur les connaissances acquises au cours du semestre : codage, notions théoriques et design patterns.

Nous avons implémenté toutes les fonctions demandées. Notre groupe étant constitué de cinq étudiants en GI02, nous avons également réalisé l'interface graphique du logiciel grâce à Qt Creator.

Nous tenons à remercier toute l'équipe enseignante de LO21 et en particulier M. Thibaud Duhautbout pour son aide tout au long du semestre, sur le projet comme en TD.

Nous décomposerons le rapport en 4 parties :

- Fonctionnalités de l'application
- Description de l'architecture et justification des choix de conception
- Capacité de l'architecture à évoluer
- Planning et contributions individuelles

NB : pour les UML qui figurent dans ce rapport, nous avons fait le choix de ne pas représenter certaines méthodes pour ne pas les surcharger.

1. Fonctionnalités de l'application

Notre application contient toutes les opérations attendues :

- ajouter/supprimer/hiérarchiser des comptes
- ajouter/corriger/supprimer des transactions simples et réparties
- calculer le solde d'un compte
- créer une clôture
- rapprocher un compte
- éditer les documents de comptabilité
- gérer la persistance des informations (fichiers, base de données, etc.)
- sauvegarder le contexte : au démarrage de l'application, l'état de l'application et les paramétrages présents lors de la dernière exécution sont récupérés.

En ce qui concerne le dernier point pour la sauvegarde du contexte, toute la base de données comprenant les comptes et les transactions est sauvegardée dans deux fichiers XML qui permettent à l'utilisateur de restaurer ces derniers à chaque nouveau lancement de l'application. Si les données de la précédentes exécution sont bien mises à jour à la relance, nous avons décidé de ne pas restaurer les vues de la session précédente car notre application se base autour d'un menu principal simple permettant d'accéder rapidement aux fonctionnalités.

En effet, notre configuration ne comporte que très peu de vues différentes, si ce n'est l'affichage des transactions ou du bilan, le reste étant des fenêtres accessibles seulement au cours d'une suite d'actions (c'est-à-dire la clôture, rapprochement d'un compte …).

Ainsi dans un souci d'intégrité et pour éviter une éventuelle perte d'informations, l'utilisateur ne peut pas reprendre une de ses actions en cours lors de la session précédente et doit par conséquent la terminer avant de fermer sa session active.

En revanche, après la fermeture de la session active, les fichiers XML utilisés pour enregistrer les données sont sauvegardés. Ce sont ces mêmes fichiers qui seront automatiquement utilisés lors de la prochaine session. Si l'utilisateur décide d'ouvrir de nouveaux fichiers de données au cours d'une session, ces fichiers-ci seront utilisés de la prochaine session. Pour faire cela, nous gérons un fichier texte qui contient les derniers

fichiers XML utilisés lors de la dernière session. Si le fichier est vide (c'est la première fois que l'application est lancée), alors les fichiers de sauvegarde sont automatiquement créés et enregistrés.

NB : toutes les opérations attendues ont été implémentées

2. Description de l'architecture et justification des choix d'architecture

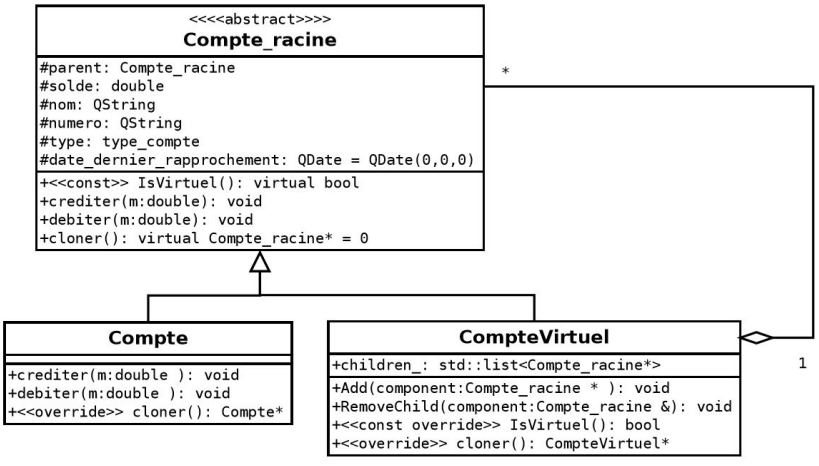
a) Compte

Architecture des comptes :

Afin de manipuler les comptes de la bonne manière, nous avons réfléchi à une architecture qui permet certaines possibilités mais restreignant également les capacités de l'utilisateur. Par exemple il fallait pouvoir réaliser des opérations bancaires (les transactions) sur certains comptes (les classiques, c'est-à-dire ceux dans lesquels nous réalisons les opérations) mais il fallait aussi empêcher l'utilisateur d'effectuer une transaction sur un compte virtuel. Ainsi, cette contrainte exigeait la création d'un type de compte ne pouvant pas contenir d'opérations mais pouvant contenir d'autres comptes comme spécifié dans le cahier des charges.
Nous avons donc cherché une architecture pouvant traiter ces spécifications et avons choisi d'implémenter le Design Pattern (DP) Composite.
Pour créer les différents types possibles nous avons implémenté une énumération :

```
enum type_compte {Actifs,Passifs,Depenses,Recettes};
```

Voici un UML simplifié de la structure des comptes :



Structure des comptes : Composite

b) Design Pattern Composite

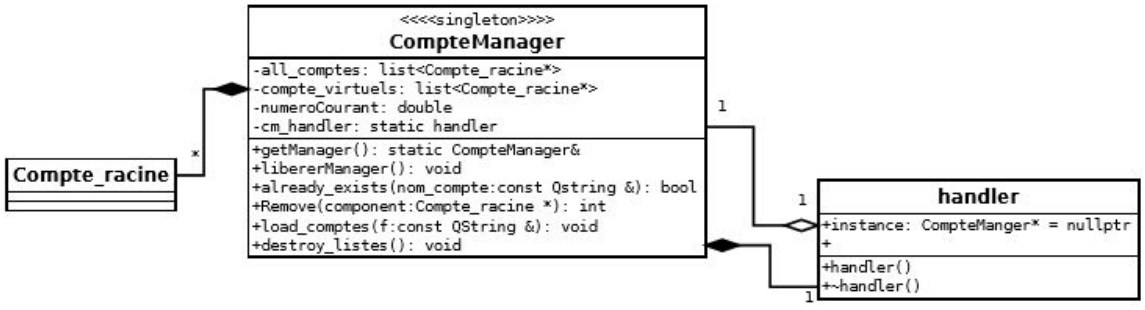
Dans notre choix conceptuel de gestion des comptes, nous choisissons d'utiliser le design pattern Composite car celui-ci permet de gérer une structure hiérarchique récursive (une arborescence de comptes).

On implémente les classes "Composant" et "Composite", Composant étant la classe mère de composite permettant de faire hériter les méthodes qui ne seront pas virtuelles pures, et qui, dans notre cas, pourra être assimilée à la classe Compte_racine.

La classe "Composite" nous permet d'organiser notre structure hiérarchique entre les comptes et les sous-comptes. La classe fille Composite intègre pleinement la création/ suppression de ses propres classes filles qui peuvent être créées par l'héritage de la méthode de création via la classe Composant.

Notre sujet impliquant la création d'un nombre indéterminé de niveaux hiérarchiques, le design pattern Composite semble donc être le design pattern le plus approprié pour gérer la hiérarchie entre les comptes de manière simple et efficace. En effet, nous pourrions également rechercher parmi tous les comptes et trouver quel compte est “contenu” dans quel compte virtuel et ainsi en déduire la hiérarchie. Nous avons pour cela mis en place un attribut *parent* pour référencer le compte virtuel qui englobe le dit compte classique. Cela nous évite de faire des fonctions de recherche coûteuses. Nous avons préféré stocker une liste de plus en mémoire plutôt que de faire des parcours consommateur en ressources. De plus, à des fins pédagogiques, l’implémentation de ce design pattern permet de manipuler l’héritage, le polymorphisme et un conteneur STL (dans notre cas les std::list, une liste chaînée).

Enfin, pour gérer le stockage de ces comptes, nous avons implémenté une classe CompteManager semblable à celle réalisée en TD mais qui possède deux listes, la liste des Compte et la liste des CompteVirtuel. Le cycle de vie des comptes sera géré grâce au DP Factory dont nous parlerons dans le paragraphe suivant.

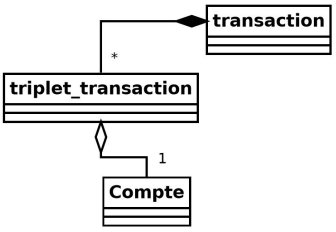


Lien entre les comptes et le CompteManager

c) Transactions

Architecture des transactions :

En ce qui concerne la structure des transactions, nous avons une classe transaction qui possède des triplets (compte, crédit, débit).
L'implémentation de triplet_transaction nous permet d'avoir des objets contenant autant de triplets que désiré : nous avons choisi de pourvoir transaction d'une std::list contenant ces triplets.



Structure des transactions

Pour regrouper, manipuler et gérer les transactions, nous avons mis à disposition une classe transactionManager semblable à celle du TD.

d) Design Pattern Singleton

Nous faisons le choix d'implémenter le design pattern singleton pour les comptes et les transactions. Nous créons deux classes Manager. Ces managers sont implémentés sous forme de singleton, c'est à dire qu'une instance d'un manager est unique, et qu'elle sera un point d'accès unique à la classe qu'ils gèrent. Ils gèrent également leurs propres objets.

Cette implémentation permet de sécuriser le code : il est bien plus simple et clair de passer par une interface, le compte manager, qui gère lui même ses objets, et qui peut-être appelé partout dans le code. Le fait qu'il gère ses objets garantit que tous les objets dont nous aurons besoin y seront stockés (dans notre cas dans des listes chaînées).

NB : Nous couplons le DP Factory au DP Singleton pour plus d'efficacité.

Au niveau de l'implémentation : on crée un compte manager que l'on implémente sous forme de singleton, qui sera récupéré avec la méthode statique getManager ailleurs dans le code. Les objets stockés dans la liste du manager sont gérés avec le DP Factory. Le compte

manager est pourvu de toutes les méthodes de recherche dans sa liste d'objets, et gère la persistance du système via les fichiers XML.

Une possibilité pour remplacer le CompteManager serait de créer des listes statiques afin qu'elles soient partagées entre tous les comptes (compte et compteVirtuel). Ainsi, n'importe quel compte aurait accès à toute la liste. Cependant cela aurait engendré la création de beaucoup de méthodes statiques (getListe, recherche dans la liste …). A l'inverse avec un compte manager, nous avons la même architecture avec uniquement une méthode statique : getManager.

e) Design Pattern Factory

Le DP factory permet de localiser en un seul endroit la création et le stockage des objets. Il est implémenté pour les comptes et les transactions. Il va de paire avec le DP Singleton des comptes managers. En effet, à chaque création de compte dans l'application, on fait appel à son constructeur. Or nous voulons centraliser le stockage des comptes dans le compte manager. C'est là que le DP factory intervient : on ajoute le clone du compte que l'on vient de créer à la liste du compte manager. Puis, toutes les opérations qui seront en lien avec ce compte feront en fait référence au clone contenu dans le compte manager. Cela garantit la solidité du code, en renforçant la centralisation des objets dans les comptes managers et l'unicité des objets dont on se sert. Ainsi tous les objets stockés , donc dans les listes, sont des clones, les autres objets seront détruits. Le principe est le même pour les transactions.

Au niveau de l'implémentation, on crée une fonction clone pour la classe qui possède un compte manager. Pour les comptes, lors de l'appel au constructeur de la dite classe, la fonction clone est appelée et c'est le duplicat résultant qui est mis dans la liste du compte manager. Ceci garantit que toute création d'un compte ou d'une transaction sera enregistrée dans les comptes managers, ainsi l'utilisateur ne pourra pas créer de compte ou de transaction "hors système" qui mettraient en péril la validité des informations dans le système (par exemple de l'argent qui disparaîtrait…). Cette manipulation permet également de simplifier le code dans les fonctions auxquelles on passe des transactions ou des comptes : il n'y a plus besoin d'un pointeur, qui peut parfois complexifier les choses, ni d'allocations dynamiques gourmandes en mémoire. On passe par référence ou on crée des variables locales, ce qui simplifiera les fonctions et la manipulation par l'utilisateur. Ainsi, le fait d'initialiser un compte (déclarer le compte et appeler son constructeur) permet également de le stocker. Pour les transactions, ce clone est créé et ajouté dans la liste des transactions lors de l'appel à la fonction ajouterTransaction qui vérifie si une transaction est correcte, la clone et l'ajoute à la liste. Il faudra par contre être prudent lors de la création de variables locales dans les fonctions : il est possible de créer des comptes temporaires qu'il faudra par la suite supprimer.

f) Conteneurs

Nous avons choisi d'utiliser un conteneur particulier, la std::list. En effet, au travers de notre expérience au cours de l'UV NF16, nous avons expérimenté l'utilisation des listes chaînées et doublement chaînées dans de nombreux cas. De plus, cela nous permettait de nous assurer de la protection de l'accès aux informations où l'on doit passer forcément par un Iterator plutôt que de passer par l'opérateur [], utilisable pour un std::vector par exemple. De plus, cela permettait de s'assurer d'avoir un conteneur dont la taille évolue dynamiquement au fur et à mesure du projet à l'aide des méthodes push_back() et push_front().

Implémentation sur QT

Qt Designer

L'implémentation de l'interface Qt s'est déroulée en plusieurs temps.

Dans un premier temps nous avons dessiné à la main l'apparence de l'interface utilisateur, pour savoir faire le lien entre l'interface et l'emploi de nos fonctions. Puis, nous avons créé la fenêtre principale à l'aide de l'outil intégré à Qt. Pour le reste, nous avons créé nos propres widgets en les faisant hériter des Widgets pré-existants. Nous avons ensuite ajouté des widgets correspondant à chaque fonction ou chaque affichage dont nous avions besoin, par exemple afficher l'arborescence des comptes et les liens de parenté.

Il a fallu remplacer les attributs des classes de notre programme "terminal" réalisé en C++ par celles de Qt, par exemple, les std::string sont devenues des QString. Nous avons fait le choix de créer, pour chaque fonctionnalité, un fichier d'en tête et un fichier de fonctions, pour décomposer le code et le rendre plus digeste, mais également plus modulaire pour ses futures évolutions.

Fenêtre principale

Le fonctionnement de notre application graphique se concentre autour d'une fenêtre principale. Cette fenêtre contient un QListWidget comme menu qui contient lui-même les

différentes fonctionnalités de l'application. Nous avons choisi une interface de ce type pour son ergonomie et sa facilité de prise en main.

Affichage des comptes :

Les comptes sont organisés en QTreeWidget, c'est-à-dire sous forme d'arborescence. Etant donnée la relation hiérarchique entre les comptes, ce Widget semblait particulièrement adapté. Le widget contient deux colonnes. La première pour le nom des comptes et la seconde pour le solde. Les éléments de l'arborescence (les comptes) sont cliquables. Il y a 4 boutons sous cette arborescence. Le bouton « Ajouter» permet, à partir du compte sélectionné, de créer un compte situé un niveau en dessous de la hiérarchie. Si le compte à créer se situe au niveau hiérarchique minimum (juste en dessous de la racine) alors on ne sélectionne aucun compte pour l'ajouter. Le second bouton permet de supprimer un compte. On s'assure que celui-ci ne contient pas de transaction s'il s'agit d'un compte « classique », ou bien qu'il ne contienne lui-même pas de compte s'il s'agit d'un compte virtuel.

Ensuite, « modifier compte » sert simplement à modifier le nom d'un compte. Enfin, le dernier bouton sert à rapprocher un compte (voir section Rapprochement bancaire).

Affichage des transactions :

Pour afficher les transactions, nous avons repris la structure vue lors du TD. Ainsi, la fenêtre transfert Viewer s'ouvre lorsque l'on clique sur "Mes transactions" dans le menu de la fenêtre principale. Cette fenêtre possède plusieurs Widgets. Tout d'abord, la liste des comptes, ainsi que le solde associé au compte qui sera sélectionné par l'utilisateur.
La sélection d'un compte dans la liste déclenche le rafraîchissement d'un QTableWidget contenant la liste des triplets pour chaque transaction concernant ce compte. Pour ce faire, nous parcourons le singleton de la classe transfertManager (historique de toutes les transactions) et pour chaque transfert, on parcourt et affiche dans le widget les informations relatives à chaque triplet correspondant au compte sélectionné.

Plusieurs opérations sur les transactions sont disponibles à partir de cette fenêtre.

- **Ajouter transaction :**

Pour ajouter une transaction, nous avons besoin de demander à l'utilisateur certaines informations. Pour cela nous avons créé une classe Information qui contiendra les informations de la transaction et qui permettra de gérer les différents cas de figure liés à l'ajout des triplets. Cette classe possède notamment un attribut "numéro" qui nous permettra de connaître le nombre de triplets déjà renseignés pour pouvoir adapter la fenêtre de saisie.
En effet, lors de l'ajout un widget apparaît pour entrer la date de la transaction ainsi que son nom (la référence est gérée automatiquement). Ensuite l'utilisateur doit appuyer sur "nouveau triplet" pour ajouter un triplet à sa transaction, il pourra alors entrer le compte, ainsi que le crédit et débit associés. Le triplet est ensuite sauvegardé et l'utilisateur peut en entrer un

nouveau ou bien confirmer la transaction si deux triplets au moins ont été saisis (d'où l'intérêt de l'attribut numéro). Une fois la transaction confirmée, on fait appel à la fonction "ajouter_transaction" dans transaction.cpp.

- **Modifier transaction :**

Le fonctionnement de "modifier transaction" est très similaire à celui de "ajouter transaction". En effet, nous avons considéré que la modification d'une transaction était en fait équivalent à une annulation suivie d'un ajout.
Ainsi, nous avons réutilisé ici la classe information.h mais en passant en paramètre du constructeur de cette classe un pointeur sur la transaction à modifier (ce paramètre, par défaut à nullptr, n'est pas utilisé pour l'ajout d'une transaction). Cela nous a permis de récupérer les informations relatives à l'ancienne transaction pour les rentrer en champs par défaut dans les widgets, ce qui est bien plus pratique et ergonomique. Une fois que la nouvelle transaction est validée, on peut annuler l'ancienne (c'est à dire faire les opérations de crédit et débit inverse).

- **Supprimer transaction**: annule la transaction et la retire de notre transactionManager.

Rapprochement bancaire et DP State

L'implémentation du Design Pattern State permet de changer rapidement le comportement d'une transaction selon son état rapproché ou non, tout en gardant une architecture propre aux transactions. Il s'agit donc d'une fonctionnalité qui s'ajoute aux transactions sans pour autant changer la nature de celles-ci.

Pour chaque transaction, l'état (rapproché ou non) est changé dynamiquement selon le contexte qui s'y prête grâce à l'attribut currentState. Ainsi, la fonction adéquate est appelée, permettant alors d'aboutir au changement de comportement désiré lors du rapprochement d'un compte. En effet, si un compte n'est pas rapproché, nous avons toujours la possibilité d'appeler les setters de la classe associée pour modifier les crédits/débits alors remplis de manière incorrecte par l'utilisateur. Cependant, grâce au design pattern state, si l'état d'une ou plusieurs transactions passe à rapproché, le comportement des setters sont modifiés de sorte qu'ils ne puissent plus modifier d'attributs pour la ou les transactions concernées, ce qui permet de sécuriser le comportement attendu pour le rapprochement.

Ce Design Pattern nous offre alors la possibilité de parcourir les transactions et d'effectuer l'ensemble des actions qui nous intéressent selon son état tout en étant assuré que les actions non désirées ne seront jamais effectuées.

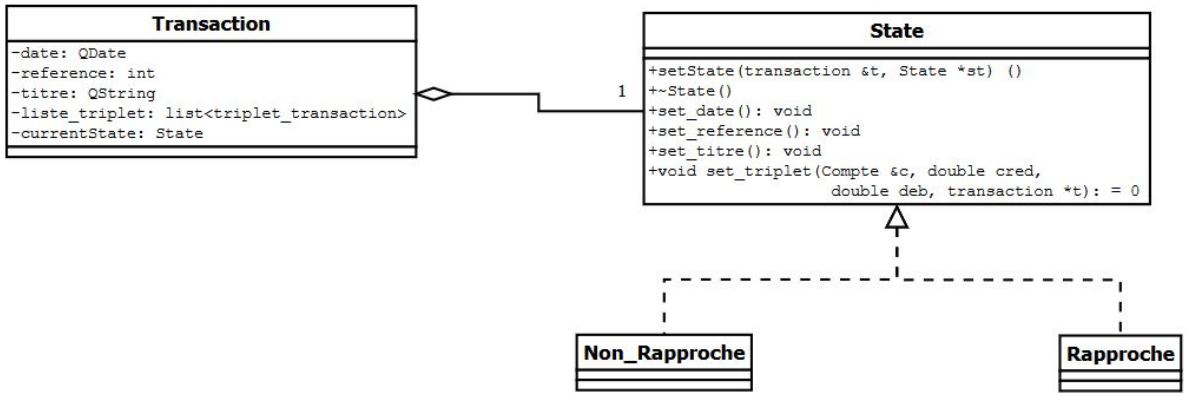


Illustration du Design Pattern State

Nous aurions pu simplement utiliser un booléen pour savoir si la transaction était rapprochée ou non. Cependant celui-ci ne permettait pas de s'adapter à une situation où l'on ajoute d'autres états. Il faudrait revenir sur la classe et ajouter des booléens par exemple. Ainsi, le DP State est plus adaptable, rien ne change dans la classe transaction, le pointeur sur State et les méthodes restent les mêmes. De plus, après une conversation avec notre chargé de TD celui-ci nous a à juste titre précisé que c'était occasion d'expérimenter le DP et que ça nous permettait d'avoir une architecture prête à l'adaptation.

Dans la fenêtre principale, si l'utilisateur sélectionne un compte non virtuel, pas encore rapproché et clique sur le bouton Rapprocher, une nouvelle fenêtre apparaît. Celle-ci contient les informations sur le solde après le dernier rapprochement et le solde actuel du compte. Si le compte n'a jamais été rapproché, la valeur affichée du solde après le dernier rapprochement prend celle du solde initial.

L'utilisateur peut alors valider ou refuser le rapprochement selon ses vérifications avec son relevé de banque.

S'il valide le rapprochement, toutes les transactions effectuées depuis le dernier rapprochement (ou depuis la création du compte si le compte n'a jamais été rapproché) jusqu'au moment du rapprochement (le jour où il le fait) passent à l'état « Rapproché » grâce au DP State. Toute modification de ces transactions devient alors impossible. Si l'utilisateur refuse le rapprochement, l'application lui propose de modifier les transactions nécessitant une correction. Toutes les transactions non rapprochées sont alors affichées et l'utilisateur peut sélectionner la référence de la transaction qu'il souhaite, sélectionner le compte, entrer le débit et le crédit des différents triplets associés à la transaction. Cette modification se fera d'abord par la suppression de l'ancienne transaction défectueuse au profit de la nouvelle transaction correcte. Lorsque toutes les corrections ont été apportées, l'utilisateur doit impérativement terminer le rapprochement du compte, c'est-à-dire que toutes les transactions

qui n'ont pas eu à être modifiées seront rapprochées. Cela est nécessaire afin d'empêcher d'avoir des transactions non rapprochées ayant une date inférieure aux transactions qui viennent d'être rapprochées. On fait donc l'hypothèse que l'utilisateur ne s'arrête pas au milieu du rapprochement mais qu'il doit bien tout vérifier.

Lien entre les différents widgets créés par nos soins

Nous avons souvent eu besoin de créer des liens entre les différents widgets. En effet, un exemple simple pour s'en rendre compte est d'imaginer que l'on crée une transaction via le widget Transaction_viewer, cette transaction va modifier le solde d'au moins deux comptes. Ainsi dans l'arborescence de la MainWindow il va falloir modifier ces soldes. Pour ce faire une fonction MAJ_listeComptes() est implémentée et appelée lorsque la transaction est créée. Pour ce faire il faut que Transaction_viewer ait connaissance de MainWindow. Nous allons donc lui fournir un pointeur sur la MainWindow pour pouvoir appeler cette méthode publique. Ici on passe même par un widget de plus pour ajouter Information. Ainsi on fait en sorte que celui-ci soit détruit quand on le ferme et sa destruction envoie un SIGNAL qui va appeler MAJ_listeComptes().

C'est en général de cette manière que l'on gère les synchronisations entre les différents widgets. Il en est de même pour la fenêtre principale qui va appeler la méthode *show() ou exec()* des différents objets lorsqu'ils doivent être affichés à l'écran. En effet une autre façon de synchroniser est de favoriser *exec()* plutôt que *show()* de façon à attendre la fin de l'exécution du widget. Il faut cependant que l'objet hérite de QDialog. Si on veut de plus empêcher la fermeture du widget car nous pouvons perdre de l'information ou corrompre certaines données, il est intéressant de redéfinir *closeEvent()*.

3. Capacités de l'architecture à évoluer

- Design Pattern State pour le rapprochement bancaire

Dans le sujet, une transaction ne peut basculer que d'un état non rapproché (état initial de toute transaction) à un état rapproché. Or, le DP State offre la possibilité de créer autant

d'états que l'on souhaite. Par conséquent, notre architecture peut s'adapter à toute évolution dans le cas où une transaction devrait rentrer dans un nouvel état. On peut par exemple penser à un nouvel état « éditable avec validation » où l'utilisateur doit valider les modifications pour les enregistrer. Il suffirait alors de créer un nouvel état dans le Design Pattern et d'inclure les fonctions nécessaires pour être certain qu'un ensemble de fonctions soit attribué à ce nouvel état. Nul besoin de modifier le squelette de transaction, ou encore de supprimer ou modifier ses méthodes.

- Design Pattern Composite pour une hiérarchie extensible

Dans le sujet, il est clairement établi que nous avons deux types de comptes, les virtuels et les non-virtuels, les comptes virtuels englobant les comptes non-virtuels comme des sous-comptes. Cependant, nous pourrions tout à fait imaginer que le client veuille étendre cette hiérarchie dans le futur en souhaitant, par exemple, implémenter un type compte hybride pouvant faire des transactions et englobant à la fois des sous-comptes. Si tel était le cas, le design pattern Composite permettrait de rajouter une troisième classe tout en y ajoutant ses spécificités localement, sans pour autant modifier toute l'architecture globale du code. En ce sens, le design pattern composite assure une hiérarchie extensible.

- Design Pattern Factory

De même le design pattern Factory reste très ouvert à l'adaptabilité. En effet, il suffit de rajouter une fonction clone pour la nouvelle classe implémentée. Sa combinaison avec Singleton garantit la solidité du code.

- Architecture QT

Extensibilité de l'interface graphique : Nous avons modélisé notre interface graphique de manière très modulaire en créant à chaque fois de nouveaux widgets dans des nouveaux fichiers dédiés. En ce sens, il serait aisé d'étendre les fonctionnalités. Graphiquement parlant, nous rajouterions de nouveaux widgets qui seraient placés à la suite des autres dans le menu des fonctionnalités. La fenêtre principale est construite pour cela.

- Typage des comptes

Pour le typage des comptes, nous avons fait le choix d'implémenter une énumération, en ce sens, si nous devions avoir une nouveau type de compte, mettons "Retraite", il suffirait juste de l'ajouter dans l'énumération.

4. Planning

Semaine	Objectif	Description de ce qui a été fait
du 13 avril au 18 avril	Première lecture du sujet en commun avec le groupe. Débuter le modèle conceptuel du projet	Mise en commun des interprétations et échanges sur les premières idées de conception Exploration des divers design patterns mentionner dans le projet et brainstorming collectif sur leur utilité face aux besoins exprimés Création d'un UML avec les classes identifiées et leurs attributs
du 20 avril au 25 avril	Réflexions en profondeur sur le modèle conceptuel et premières implémentations	Choix de plusieurs design patterns (découverte de Composite et State) Finalisation d'un premier modèle UML et soumission au chargé de TD pour revue Première implémentation sous Code Blocks pour le design pattern Composite. Réflexion sur les divers conteneurs qui seront utilisés
du 27 avril au 02 mai	Pivoter ou non en fonction du retour du chargé de TD Concentration du groupe sur l'implémentation sous CodeBlocks	Prise en compte du retour du chargé de TD Répartition des tâches entre les membres du groupes par partie du projets: - Exploration de Composite - Découverte & tentatives sur State - Classes & méthodes simples (créditer/débiter, calcul du solde, ajout de triplets....)
		Répartition des membres du groupe sur la réalisation des fonctions suivantes: - ajout/suppression/modification

du 04 mai au 09 mai	Concentration sur l'implémentation des fonctions clés du projet	transaction - ajout/suppression/modification compte - clôture d'un compte - rapprochement d'un compte
du 11 mai au 16 mai	Revue et correction des différentes fonctions implémentées par chacun des membres du groupe Familiarisation avec l'environnement Qt	Correction et finalisation des tests sur les fonctions réalisées sous CodeBlocks Téléchargement de Qt en Opensource par tous les membres du groupe, relecture du TD6 sur Qt et brainstorming sur l'interface graphique
du 18 mai au 23 mai	Qtéisation de tous nos attributs et méthodes/fonctions Création des widgets propres à notre interface et linkage avec nos fonctions	Qtéisation de nos attributs et méthodes/fonctions Réutilisation de la philosophie du TD6 pour implémenter une fenêtre de vue sur les transactions Découverte et utilisation du QTreeWidget pour l'arborescence
du 25 mai au 30 mai	Qtéisation de tous nos attributs et méthodes/fonctions Finalisation et tests sur l'arborescence et la fenêtre de transaction Gérer la persistance des informations (format XML)	Qtéisation spécifique sur la clôture, l'ajout, la modification et la suppression de transactions. Lecture de la documentation fournie sur le stockage de données via XML, implémentation sous Qt et premiers tests avec les transactions et les comptes.
du 01 juin au 06 juin	Qtéisation du rapprochement Editer les documents de comptabilité Gérer la persistance des données	Finalisation de la gestion de la persistance des données Premier document de comptabilité édité : bilan Implémentation approfondie du rapprochement, qtéisation et linkage

	Remise en question de l'architecture des comptes et gestion de la mémoire	avec de nombreux widgets. Ajout d'un compteManager à la place de liste statiques, implémentation de Factory Method
du 08 juin au 13 juin	Finaliser l'édition des documents de comptabilité Finaliser le rapprochement Sauvegarde du contexte Revue de nombreuses fonctions & débuggage Début de rédaction du rapport	Tous les documents de comptabilité ont été édité : bilan & relevé des recettes et dépenses. Implémentation d'un historique de toutes les transactions, peu importe le compte. Implémentation de la sauvegarde du contexte Correction de nombreux bugs Rédaction de l'introduction, fonctionnalités produit et détails sur l'architecture
du 15 juin au 18 juin	Revue de toutes les fonctions Réaliser une multitude de tests Vérifier indentation & pertinence des commentaires Finalisation du rapport & doxygen Tournage de la vidéo	Correction de nombreux bugs Réalisation de divers tests spécifiques aux fonctions clés du projet Version Finale du Rapport Version Finale du code Version Finale doxygen Vidéo tournée

NB : Tout au long du projet, beaucoup de temps à été consacré à mettre en commun les codes des différents membres du groupe et à modifier les fonctions selon les bugs, la prise en compte des exceptions, la complexité etc.

5. Contributions personnelles

Commun : Lire et décortiquer le sujet, réflexion théorique et création du l'UML, Réunions hebdomadaires (2 à 3 fois par semaine, 4 heures le Jeudi après-midi et sessions le week-end) communiquer avec les enseignants, débuggage, rassemblement des codes, écriture du rapport, multiples tests sous QT

Ilan : Implémentation du DP State avec Sébastien, implémentation des fonctions de transactions, fenêtre transaction_viewer.h, persistance des données (XML write & load avec Sébastien), Edition des documents de comptabilité: bilan/Compte de résultats (création des widgets:compte_resultat_viewer.h,bilan_viewer.h avec formulaire_date.h pour la période donnée, ainsi que toute l'algorithmie associée dans les widgets), Création de l'historique des transactions (all_transactions.h), historique des rapprochements (all_rapprochements.h), fonction rapprocher avec Sébastien. Gestion de l'annulation d'actions (débuggage)

Hippolyte : Implémentation du DP Composite pour les comptes, croquis de l'interface graphique, création de la mainWindow, signaux des boutons relatifs aux compte et transaction (ajouter, supprimer,modifier), création et modification avec Julien des fonctions relatives aux transactions et adaptation de information.cpp pour modifier_transaction.
Création de la classe fichiersXML.cpp pour gérer la persistance des données et fonctions liées à la sauvegarde et à l'ouverture de nouveaux fichiers de données.

Julien : Implémentation DP Composite, Factory Method, Singleton (transactionManager). Création/modification des widgets suivants : Capitaux_propres qui gère les soldes initiaux, choix_type qui permet de choisir le type d'un compte qui n'est pas dans un compte virtuel, compte_resultat_viewer (modification et corrections de bugs), création du widget information qui permet de renseigner les différentes données d'une transaction, implémentation de fonctions de MainWindow, correction de bugs du rapprochement bancaire, correction de bugs de transaction_viewer. En particulier gestion des liens entre les différents widgets (refresh des affichages etc). Implémentation de fonctions de compte, transaction.
Mise en commun du code, intégration des différentes implémentations et aide à la décision.

Kilian : Recherche des DP, implémentation des singletons, des managers, de factory method sous Qt & gestion des xml sous-jacente, croquis interface graphique, fonction de clôture du livre, mise en commun des codes avec Manager et Factory, diverses fonctions liées aux comptes et à compte / transaction manager, QTreeWidget et affichage de l'arborescence dans MainWindow, création des comptes sans père, fonctionnement et logique des capitaux propres, documentation Doxygen.

Sébastien : Implémentation du Design Pattern State avec Ilan, de l'opération Rapprochement bancaire (fonctions et Qt) et modifications en conséquence des fonctions d'ajout/modification de transactions selon les contraintes de rapprochement, modification de la fenêtre principale et par conséquent du fichier MainWindow.cpp. Participation à l'implémentation des fonctions liées à l'XML avec Ilan. Participation de l'élaboration du QTreeWidget pour la création de comptes issus d'aucun compte parent. Désallocation des fonctions. Gestion de l'annulation d'actions (Corrections de bugs sur les boutons Cancel ou la croix [X] d'une fenêtre)

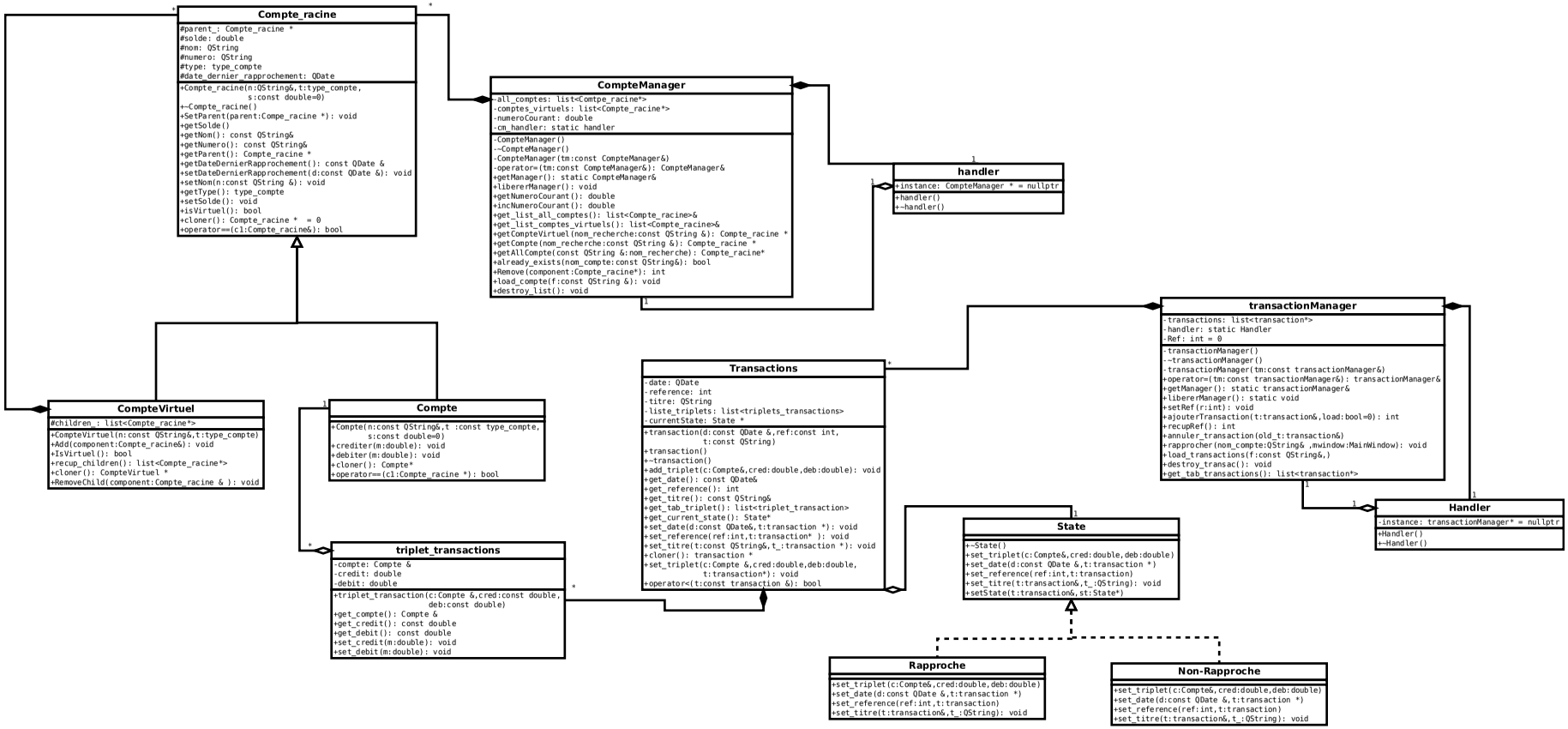
Conclusion

Ce projet, ainsi que l'UV LO21 dans sa globalité, nous aura permis de découvrir de manière concrète ce qu'est la programmation orientée objet. Nous avons apprécié chercher et implémenter les design patterns vus en cours, ou bien trouvés via nos recherches personnelles. Sortir du cadre théorique et se rendre compte que nous avons un souci dans le code, soluble par un DP a été très formateur et satisfaisant. Nous avons également découvert Qt creator, que nous pourrions être amenés à réutiliser dans le futur.

Apprendre à utiliser les widgets et débugger le code aura été la partie la plus fastidieuse et longue du projet, mais nous pensons que c'est la philosophie UTC : apprendre à apprendre et à se débrouiller par nous mêmes. Nous n'en sortirons que grandis, et plus opérationnels une fois en TN09. Pour certains membres du groupe, ce projet aura aussi été une bonne introduction au travail de groupe complexe : nous devions être organisés et mettre fréquemment les idées et les codes en commun pour pouvoir rester cohérents entre nous ; les réunions d'équipe et la bonne entente au sein du groupe ont été essentielles au succès de ce projet.

Nous tenons à remercier l'équipe enseignante pour sa disponibilité et sa pédagogie, particulièrement M. Antoine JOUGLET et M. Thibaud DUHAUTBOUT avec qui nous avons eu le plus de contacts, surtout au vu des circonstances actuelles : LO21 a été l'une des UVs les plus agréables à suivre.

Annexes


Compte_racine
#parent_: Compte_racine *
#solde: double
#nom: QString
#numero: QString
#type: type_compte
#date_dernier_rapprochement: QDate
+Compte_racine(n:QString&,t:type_compte, s:const double=0)
+~Compte_racine()
+SetParent(parent:Compe_racine *): void
+getSolde()
+getNom(): const QString&
+getNumero(): const QString&
+getParent(): Compte_racine *
+getDateDernierRapprochement(): const QDate &
+setDateDernierRapprochement(d:const QDate &): void
+setNom(n:const QString &): void
+getType(): type_compte
+setSolde(): void
+isVirtuel(): bool
+cloner(): Compte_racine * = 0
+operator==(c1:Compte_racine&): bool
CompteManager
-all_comptes: list<Comtpe_racine*>
-comptes_virtuels: list<Compte_racine*>
-numeroCourant: double
-cm_handler: static handler
-CompteManager()
-~CompteManager()
-CompteManager(tm:const CompteManager&)
-operator=(tm:const CompteManager&): CompteManager&
+getManager(): static CompteManager&
+libererManager(): void
+getNumeroCourant(): double
+incNumeroCourant(): double
+get_list_all_comptes(): list<Compte_racine>&
+get_list_comptes_virtuels(): list<Compte_racine>&
+getCompteVirtuel(nom_recherche:const QString &): Compte_racine *
+getCompte(nom_recherche:const QString &): Compte_racine *
+getAllCompte(const QString &:nom_recherche): Compte_racine*
+already_exists(nom_compte:const QString&): bool
+Remove(component:Compte_racine*): int
+load_compte(f:const QString &): void
+destroy_list(): void
handler
+instance: CompteManager * = nullptr
+handler()
+~handler()
CompteVirtuel
#children_: list<Compte_racine*>
+CompteVirtuel(n:const QString&,t:type_compte)
+Add(component:Compte_racine&): void
+IsVirtuel(): bool
+recup_children(): list<Compte_racine*>
+cloner(): CompteVirtuel *
+RemoveChild(component:Compte_racine &): void
Compte
+Compte(n:const QString&,t :const type_compte, s:const double=0)
+crediter(m:double): void
+debiter(m:double): void
+cloner(): Compte*
+operator==(c1:Compte_racine *): bool
triplet_transactions
-compte: Compte &
-credit: double
-debit: double
+triplet_transaction(c:Compte &,cred:const double, deb:const double)
+get_compte(): Compte &
+get_credit(): const double
+get_debit(): const double
+set_credit(m:double): void
+set_debit(m:double): void
Transactions
-date: QDate
-reference: int
-titre: QString
-liste_triplets: list<triplets_transactions>
-currentState: State *
+transaction(d:const QDate &,ref:const int, t:const QString)
+transaction()
+~transaction()
+add_triplet(c:Compte&,cred:double,deb:double): void
+get_date(): const QDate&
+get_reference(): int
+get_titre(): const QString&
+get_tab_triplet(): list<triplet_transaction>
+get_current_state(): State*
+set_date(d:const QDate&,t:transaction *): void
+set_reference(ref:int,t:transaction*): void
+set_titre(t:const QString&,t_:transaction *): void
+cloner(): transaction *
+set_triplet(c:Compte &,cred:double,deb:double, t:transaction*): void
+operator<(t:const transaction &): bool
transactionManager
-transactions: list<transaction*>
-handler: static Handler
-Ref: int = 0
-transactionManager()
-~transactionManager()
-transactionManager(tm:const transactionManager&)
+operator=(tm:const transactionManager&): transactionManager&
+getManager(): static transactionManager&
+libererManager(): static void
+setRef(r:int): void
+ajouterTransaction(t:transaction&,load:bool=0): int
+recupRef(): int
+annuler_transaction(old_t:transaction&)
+rapprocher(nom_compte:QString& ,mwindow:MainWindow): void
+load_transactions(f:const QString&,)
+destroy_transac(): void
+get_tab_transactions(): list<transaction*>
Handler
-instance: transactionManager* = nullptr
+Handler()
+~Handler()
State
+~State()
+set_triplet(c:Compte&,cred:double,deb:double)
+set_date(d:const QDate &,t:transaction *)
+set_reference(ref:int,t:transaction)
+set_titre(t:transaction&,t_:QString): void
+setState(t:transaction&,st:State*)
Rapproche
+set_triplet(c:Compte&,cred:double,deb:double)
+set_date(d:const QDate &,t:transaction *)
+set_reference(ref:int,t:transaction)
+set_titre(t:transaction&,t_:QString): void
Non-Rapproche
+set_triplet(c:Compte&,cred:double,deb:double)
+set_date(d:const QDate &,t:transaction *)
+set_reference(ref:int,t:transaction)
+set_titre(t:transaction&,t_:QString): void