LO21 : Projet trésorerie

Intégrants :

- **AMBROZIAK Sébastien**
- **CHEN Ruochen**
- **ZUNIGA Uriel**
- **Oumnia Ait El Hihi**

Table de metieres

Résumé de l’application

Notre programme permet :

- Ajouter des comptes
- Supprimer des comptes
- Hiérarchiser des comptes
- Ajouter Transactions
- Supprimer Transactions
- Modifier Transactions
- Créer clôture
- Rapprocher un compte
- Éditer les documents de compatibilité
- Gérer la persistance d’information
- Sauvegarder le contexte

Le programme est initialisé à son premier démarrage avec un compte “Racine” uniquement. Ce compte constitue la base de notre arborescence auquel nous pouvons ajouter des comptes virtuels ou réels à l'aide du bouton “Add Account” dans le menu edit. Il est possible d’ouvrir de nouvelles fenêtres qui nous aideront à démarrer une nouvelle instance du programme.

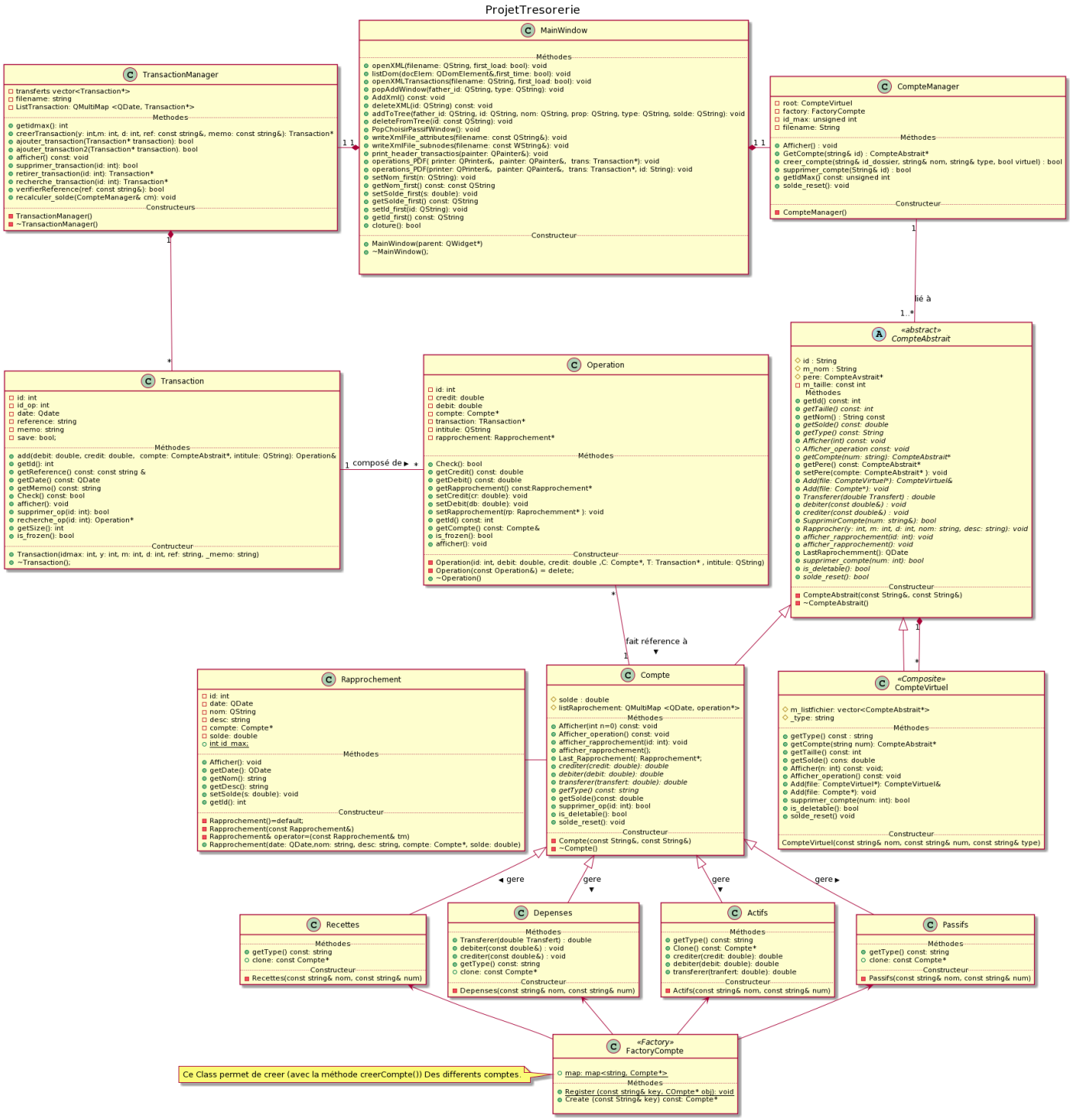
Il est possible de lire deux fichiers xml au début du programme grâce au boutons file/Open, un pour les comptes et un pour les transactions. Cela nous aidera à poursuivre notre travail plus tard sans perdre aucune information. Chacune des lectures des fichiers se fait avec différentes fonctions puisque les fichiers xml ont une structure différente pour sauvegarder le contexte. Nous avons été prudents avec les fonctions d'ajout de Transferts dans le transactionManager car si lors du chargement des fichiers, nous ajoutons à nouveau les transactions, nous modifierions les soldes réels des comptes associés, pour eux, une fonction a été créée qui ajoute des transactions au TransfertManager sans modifier le solde.

Le programme permet donc de créer et modifier et supprimer des transaction entre plusieurs compte à l’aide des boutons implémentés dans le menu Edit. Il est possible d’afficher toutes transactions ou uniquement celles d’un compte dans l’interface à l’aide du menu Show. Le menu Edit permet aussi de rapprocher un compte à une date sélectionné et d’effectuer des clôtures de compte. Enfin il est possible de générer des rapports pdf et de sauvegarder le contexte de l’application en sauvegardant le fichier xml des comptes et des transferts.

Les fonctions qui n’ont pu être implémentées sont :

- Génération des des rapport pdf des rapprochements
- Sauvegarde de contexte des rapprochement (enregistrement sous forme de fichier xml non implémenté)

Architecture



Img 1.1 Diagramme des classes

Architecture des classes :

Compte :

Concernant la gestion des comptes et des transactions, nous étions partis de l'idée que les classes “TransfertManager” et “CompteManager” devaient avoir un design pattern iterator Singleton mais peu de temps après, lors du développement du programme dans Qt, nous avons créé une fonction qui s'initialise et nous permet de travailler avec deux fenêtres en même temps, nous avons donc décidé qu'il était préférable de supprimer ces conceptions de notre architecture.

Afin de parcourir les listes, la liste QMultiMap et d'autres structures de données, nous préférons normalement que les classes héritent directement de ces structures de données afin que nous puissions facilement utiliser des méthodes comme begin () ou end () de cette façon, nous pouvons facilement connaître le début et la fin de nos structures.

Le logiciel nécessitant de gérer les comptes avec une arborescence, après consultation des différents designs pattern existants nous nous sommes orientés vers le design pattern **Composite.** En effet, ce design pattern permet une représentation de dossiers et sous dossiers ainsi que des fichiers, cette structure s’adapte donc parfaitement à la gestion de compte demandé en représentant un “compte réel” comme un fichier et un “compte virtuel” comme un dossier. Le compte racine ainsi que les différents types de comptes réels et virtuels peuvent être implémentés. Comme le montre notre diagramme de classe, nous avons implémenté les quatres types de compte réel en les faisant hériter de la classe Compte (représentant un compte réel et un fichier de base dans notre composite).

Nous n’avons pas fait de comptes virtuels particuliers par héritages car ils ne requièrent pas de fonctionnement particulier, seul un attribut “_type” à été implémenté, il permet de contenir le type d’un compte en chaine de caractère, dans notre cas pour les compte virtuels nous utiliserons que les type (“Racine”,”Actif”,”Passif”,”Depense”,”Recette”) si nécessaire le type peut être testé dans les fonctions pour attribuer des comportement particuliers à ces comptes, mais cela n’est pas souvent nécessaire dans notre cas. Nous avons donc préféré garder une seule classe compte virtuel plutôt et de tester quand nécessaire le type du compte au lieu de de faire 5 héritages pour des dossiers qui fonctionnent de la même manière. Concernant les Comptes réels, la classe Compte a été implémentée, représentant un compte réel sans type permettant de définir les méthodes communes de tous les comptes réels. Les comptes de différents types ont été implémentés par héritage pour permettre de modifier leurs fonctionnements au niveau des opérations (inversées pour les comptes de dépenses et d’actifs). Un des inconvénients de cet héritage est que pour créer un compte réel, il faut appeler le constructeur d’un type de compte ce qui peut rajouter de la complexité notamment dans la création des comptes, tel quel, il faut faire un “case” ou 4 “if” pour créer un compte du type choisi à partir d’une seule fonction.

Cependant, nous avons implémenté le design pattern **factory** pour palier à ce problème. Le design pattern Factory, ou **Fabrique** est un design pattern permettant de

séparer la création d'objets dérivant d'une classe mère de leur utilisation. De ce fait, on a alors la possibilité de créer plusieurs objets issus d'une même classe mère. Nous nous sommes inspirés sur un modèle de “Toys” qui suit le même principe que les différents types de comptes.

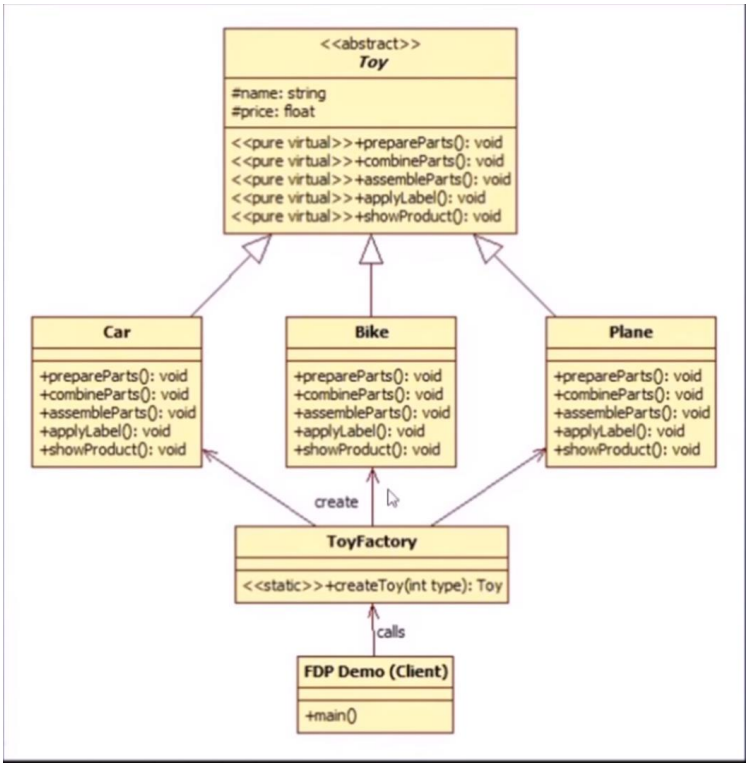


Image 1.2 Design Pattern “factory” Toys

On a suivi alors le même principe que les “Toys” pour generer differents types de Comptes réels. Alors avec une méthode clone() on va créer les différents types de Comptes (À partir du constructeur) et la Class factory va créer avec une même méthode n’importe quel type de compte. L’utilisation de ce design permettra à l'avenir d'incorporer différents types de comptes simplement en modifiant certains paramètres. En plus de la création de comptes est plus confortable au niveau de la programmation

Transaction :

En accord avec le sujet, l'architecture des transactions est basée sur les classes :

- Opération
- Transaction
- TransactionManager

Ces trois classes ont été implémentées afin de sauvegarder toutes les informations nécessaires aux transactions et de manière modulable.

Opération :

Une Opération représente un débit ou un crédit sur un seul compte, elle a pour seul but de retenir une action demandée sur le solde d'un compte. Une Opération contient un id, les montants débit et crédit, un pointeur vers le compte sur lequel elle agit, un pointeur vers sa transaction un intitulé et enfin un pointeur vers son rapprochement initialisé à nullptr.

Transaction :

Une transaction est un conteneur d'opérations et représente donc des opérations sur plusieurs comptes, ce qui permet de faire des transferts. Cette classe dispose du design pattern iterator pour lister ses opérations. La classe transaction est responsable de la création des opérations (le constructeur d'opération est privé et transaction est déclaré comme ami). Cette classe contient un id attribué par le manager de transaction, un id compteur à attribuer à ses opérations (clé locale), une date, une référence, un descriptif et un vecteur d'opération.

TransactionManager

Les transactions sont gérées et stockées par le TransactionManager. Celui-ci à pour but de créer et d'attribuer l'id des transactions, cette classe possède aussi un design pattern iterator afin d'enregistrer les transactions. Cette classe permet aussi d'appliquer les transactions aux comptes avec la méthode ajouter_transaction() qui enregistre une transaction dans le dictionnaire et qui applique les débits et crédits à chaque compte de la transaction. Cette manière de procéder permet à l'interface de créer des transactions temporaires avant de les enregistrer. De la même manière, il est possible de modifier une transaction avec la méthode "retirer_transaction()" qui à pour but de retirer une transaction du dictionnaire et d'annuler les opérations sur les comptes afin de pouvoir la modifier et la ré enregistrer plus tard. Ce manager permet aussi de supprimer une transaction qui elle-même va supprimer ses opérations. Les opérations se suppriment du vecteur de leurs compte grâce à leurs destructeurs.

Rapprochement :

Un Rapprochement tout commes les autres classes de transactions est un conteneur d'informations. Il contient un id, une date, un nom, une description, un pointeur vers le compte rapproché et une sauvegarde du solde au moment du rapprochement.

Architecture Compte/Transaction.

Afin de relier la partie transaction avec les comptes, nous avons commencé par implémenter le design pattern iterator sur la classe "Compte". afin d'accéder facilement aux opérations d'un compte sans avoir à parcourir toutes les transactions existantes, nous avons implémenté un QMultiMap<QDate, Operation*> dans la classe Compte. Le multimap avec une date d'opération pour clé permet de trier par date les opérations.

Le fait de pouvoir accéder à la liste des opérations depuis un compte permet de pouvoir accéder aussi à ses transactions, en effet, chaque opération possède un pointeur vers sa transaction, cela a eu pour effet de faciliter l'accès aux transactions d'un compte dans l'interface.

Le rapprochement d'un compte se fait depuis le compte grâce à la méthode Compte::rapprocher(), cette méthode créer un rapprochement et initialise le pointeur de rapprochement de chaque opération non rapprochée en dessous de la date indiquée. Chaque rapprochement est enregistré dans le Multimap "ListRpprochement" de la classe Compte. Cette structure a été pensé pour permettre à l'interface de lister les rapprochements d'un compte facilement avec l'itérateur, mais aussi d'afficher les opérations liées à un rapprochement en accédant à l'itérateur d'opération du compte et en comparant le pointeur de rapprochement des opérations à celui du rapprochement désiré.

Architecture de l'interface :

Main Window

Cette classe vous permet de gérer la fenêtre principale, en son sein, nous avons certains attributs tels que le CompteManager et leTransfertManager qui serviront toujours à modifier l'état actuel du programme. D'autre part, il existe des méthodes qui permettent d'ouvrir de nouvelles fenêtres, de lire des fichiers xml, de générer des pdf entre autres
Dans la documentation, il y a une description plus détaillée du fonctionnement de cette classe. De la même manière, la vidéo et les annexes expliquent le fonctionnement des slots pour effectuer des actions lorsque l'on clique sur un bouton.

Add Compte Dialog

Cette classe vous permet de gérer la vente qui ajoute des comptes dans l'interface. Comme la classe précédente, les méthodes sont mieux expliquées dans la documentation.

Add Transfert Dialog

Cette classe vous permet de gérer la vente qui ajoute des transactions dans l'interface. Comme la classe précédente, les méthodes sont mieux expliquées dans la documentation.

Choisir Passif Dialog

Cette classe permet d'ouvrir une fenêtre pour sélectionner un compte passif lors de la création d'un compte actif

Rapprochement Dialog

Cette classe nous permet de créer un rapprochement de compte. Au début, une fenêtre s'ouvre qui nous montrera la date, le nom et la description, nous devons remplir ces informations et enfin à l'aide d'un bouton, la méthode de la classe de compte sera appelée

Saisir Rapprochement Dialog

Dans un premier temps il prend un compte et à partir de là il parcourt sa liste de rapprochements pour pouvoir laisser l'utilisateur en choisir un. Dans la version finale, cette fonction n'était pas terminée.

Evolutivité du logiciel

Composite

Design pattern flexible permettant de créer des arborescences comme on le souhaite, il est possible grâce à ce design pattern de stocker des objets de différents types dans le même arbre, cela pourrait permettre par la suite de créer de nouveau type fichiers avec différents comportements qui seront compatible avec l'arborescence.

Factory

La factory permet dans notre cas de faire abstraction du nombre de classes héritées Compte, donc de type de "compte réel" ce qui permettrait d'ajouter des nouveaux comptes avec leurs propre fonctionnement très facilement. La procédure pour créer un nouveau type de compte serait de faire un nouveau compte hérité de la classe "Compte", d'y implémenter une méthode clone() et de l'enregistrer dans le dictionnaire de la factory.

Polymorphisme (classes comptes)

Complété avec la factory, le polymorphisme permet de créer autant de type de comptes que souhaité avec leurs propres méthodes.Le polymorphisme permet aussi de garder une structure de base de notre fichier(compte réel) avec des attribu et des méthode utilisable par n'importe lequel des comptes particuliers hérités de la classe Compte. La combinaison du design pattern factory avec le polymorphisme des fichiers semble donc être une architecture très évolutive permettant d'ajouter à n'importe quel moment des nouveau types de fichiers sans impacter le reste du logiciel.

Séparation frontend/backend (permet de faire évoluer les 2 indépendamment)

Le projet a été fait en équipe et donc les partie front-end et back-end ont été séparés. L'Objectif pour la partie back end était donc de fournir des classes et une architecture facile d'utilisation pour une implémentation front-end au niveau de l'interface tout en réfléchissant à la complexité des fonctions. Le fait d'avoir développé le logiciel de cette manière permet d'avoir une base solide adapté à n'importe quelle interface. Cette séparation permet aussi de modifier le back-end ou le front end à tout moment.

Modularité des transactions

L'utilisation de pointeurs dans les opérations et transactions permettent d'accéder facilement aux informations. à partir d'une opération, il est possible d'accéder à son compte , à sa transaction et à son rapprochement peu importe le type du compte réel. De même que les opérations, il est possible à partir d'un compte d'accéder à toutes ses opérations et ses rapprochements, Toujours grâce au polymorphisme, il est possible de créer des transactions et des rapprochements pour n'importe quelle classe héritée de Compte, et donc tout nouveau compte implémenté pourrait utiliser ce système.

Beaucoup d'associations entre les classes.

Comme nous l'avons vu avec les transactions, le fait d'avoir utilisé beaucoup de pointeurs entre les classes principales permet d'accéder facilement aux classes de communiquer et d'accéder aux informations. Cela permet donc d'intégrer facilement au système des nouveaux objets par héritages des classes principales.

Les comptes gèrent leurs opérations (possibilité d'implémenter les transactions sans manager)

En effet, il serait possible grâce à ses liaisons de se passer du transfert manager étant donné que les comptes peuvent gérer leurs informations, on pourrait aussi se passer du compte manager grâce à l'arborescence, simplement en utilisant le compte racine. Il serait donc possible de refaire l'architecture de la gestion des comptes en implémentant les fonctions du compte manager dans l'interface par exemple. Nous avons opté pour l'utilisation de manager afin de faciliter le travail de groupe et de fournir une interface simple à utiliser entre les classes et l'interface logiciel. Les manager permettent donc de faire abstraction de la structure réelle des classes et ainsi simplifier le travail d'équipe.

Système de fichier séparé

Tout comme la partie logiciel et classe, le système de fichier a été séparé et utilise différents itérateurs fournis par les classes principales pour accéder aux informations efficacement en évitant d'augmenter la complexité. Le système de fichier à pu utiliser aussi les managers afin d'utiliser des méthodes déjà implémentés. Cette partie n'a nécessité aucune modification du logiciel, ce qui est aussi pratique pour de futures évolutions, il serait possible par exemple de changer entièrement le système de sauvegarde de contexte sans avoir à changer les classes ou l'interface du logiciel.

Planning

Task / Date	8/4/2020	15/4/2020	22/4/2020	29/4/2020	6/5/2020	13/5/2020	20/5/2020	27/5/2020	3/6/2020	10/6/2020	17/6/2020
Analyse du sujet											
Définition de l'architecture											
Création du diagramme UML											
Création class Transaction											
Création class TransactionManager											
Création class Compte Manager											
Création des differents class de comptes											
Developpement interface QT											
Union de transfert de classe et compte											
Création de menu dans la console											
Utilisation des comptes classes et des transactions dans Qt											
Lecture / Ecriture des fichiers XML											
Rapprochement des comptes											
Clôture de Comptes											
Solutions des problemes sur l'interface											
Géneration des rapports (PDF)											
Test complet du programme											
Rapport écrit											
Video											
Documentation Doxygen											

Au début, c'était la planification du projet, mais il est vrai que parfois certaines tâches prenaient plus de temps ou lors de la mise en place de certaines classes, il y avait des problèmes et nous devions revenir un peu en arrière pour résoudre ces problèmes. L'objectif était de pouvoir arriver à la semaine dernière avec le projet terminé et dans le pire des cas ne corriger que quelques bugs d'utilisation.

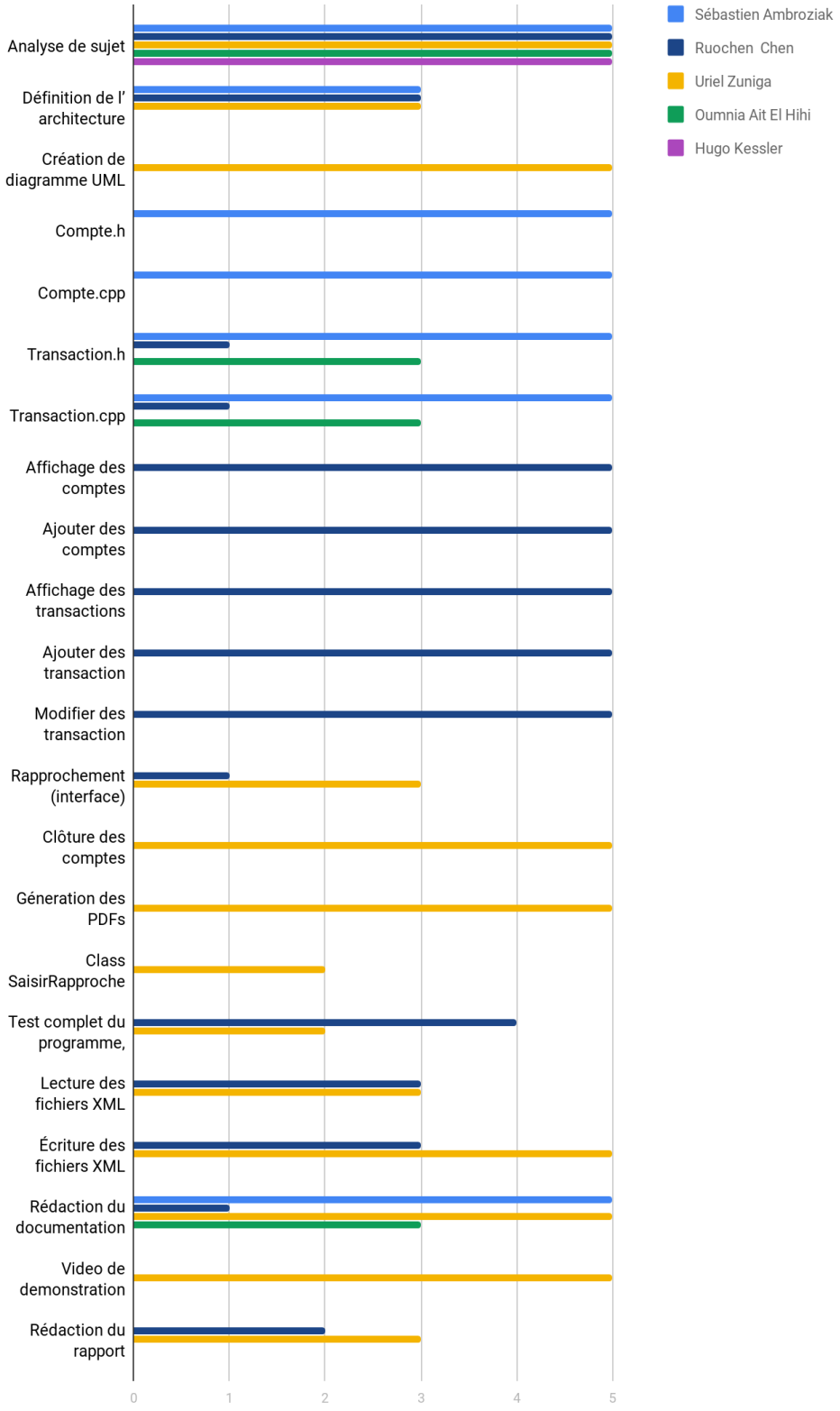
Certaines tâches en avaient besoin d'une autre pour démarrer, par exemple nous ne pourrions pas commencer à définir l'architecture du programme si nous n'avions pas lu tous les problèmes présentés.

Une fois l'architecture que nous utiliserons définie, nous avons commencé avec la partie de la programmation des classes qui serait la base du fonctionnement du projet. À l'occasion, certains designs patterns ont été ajoutés pour aider à améliorer le fonctionnement futur du programme. Dans le même temps, nous avons commencé par concevoir une interface graphique sur Qt qui, à l'avenir, lorsque la classe des transactions et des comptes fonctionnerait ensemble, ils s'adapteront à l'interface pour pouvoir utiliser dynamiquement différentes fonctions.

Comme déjà mentionné, à certaines occasions, nous avons même remarqué l'implémentation de l'interface que nous avons dû ajouter ou modifier certaines méthodes de classe pour mieux les adapter au projet final avec l'interface graphique.

Contribution - diagramme de bande

Remarque: 5 signifie une contribution maximum



Description détaillée :

Sébastien Ambroziak:

Ma contribution au projet à consister en l'analyse du sujet afin de proposer et d'implémenter un système d'arborescence de compte en accord avec le sujet. Mon choix c'est donc porté sur le design pattern composite. J'ai implémenté par la suite les différents types de comptes et choisi d'implémenter du polymorphisme uniquement pour les comptes réels. Pour tester la partie compte j'ai créé un menu afin de tester mes classes et de débuter le projet console en même temps. J'ai par la suite implémenté le design pattern factory pour faciliter la création de comptes réels dans le menu mais aussi pour faciliter la création dans le logiciel final. J'ai par la suite contribué au projet en faisant une refonte de l'architecture des transactions notamment en ajoutant la classe transaction afin d'en faire un conteneur d'opérations, en modifiant la classe opération notamment en ajoutant des pointeurs afin de faire des liaisons entre les comptes, les transaction les opérations et les itérateurs. J'ai contribué également à l'élaboration du transactions manager ainsi que ses méthodes permettant et facilitant la création, modification, suppressions et enregistrement des transactions en accord avec les demandes du logiciel et de mes collègues se chargeant de toute la partie interface et sauvegarde de contexte. J'ai aussi contribué à la partie rapprochement de compte en ajoutant les méthodes et les classes nécessaires au niveau du back-end.

- Ajouter/supprimer/hiérarchiser des comptes (back-end)
- Ajouter/corriger/supprimer des transactions simples et réparties (back-end)
- Calculer le solde d'un compte
- Rapprocher un compte (back-end)
- Version console des comptes/transactions
- Rédaction documentation doxygen (Partie Compte et méthodes de transactions.cpp)
- Contribution au rapport avec mes camarades : Architecture des classes et évolutivité du logiciel.

Ruochen Chen:

Je m'occupe principalement la partie interface et je gère aussi connexion des classes ComptesManager et TransfertsManager avec le Qt. Ceci consiste à lire les comptes dans un fichier xml et les afficher dans le qtreewidget avec une structure hiérarchique, et de réécrire les informations des comptes dans un fichier xml. Puis, j'ai réalisé la fenêtre pour créer un nouveau compte, et choisir le compte passif pour initialiser les comptes si nécessaire. Si le dernier n'existe pas, alors l'application va lui proposer de créer un nouveau compte passif. Ensuite, pour la création d'une transaction, j'ai conçu une fenêtre pour saisir toutes les informations nécessaires, puis le créer dans le TransactionManager. S'il y a un problème lors de saisie (par exemple, la somme de débit n'est pas égale à la somme de crédit, ou un champ vide), alors un QMessageBox va s'ouvrir pour lui informer le problème. De plus, j'ai créé une fenêtre pour afficher toutes les transactions, elles sont listées dans un qTableWidget dans cette fenêtre. En même temps, on peut cliquer sur le bouton "modify" pour modifier une transaction en particulier. L'application s'affiche alors la même fenêtre pour créer une transaction. Finalement, je participe aussi au test complète de l'application.

Uriel Zuniga :

Surtout ma tâche a eu plus d'effet au niveau du projet Qt. Lorsque nous avons commencé à ajouter les différentes classes dans le projet final, nous avons commencé à rencontrer des défis tels que l'enregistrement du contexte chaque fois que l'utilisateur le souhaitait, la gestion de différents objets dans qt (comme des QTable, QTreeWidget, QDomElement, ...) pour afficher les états de compte et de transaction. Par exemple dans le cas de comptes réels avec un rapprochement, on peut mettre en évidence la couleur des transactions en rouge. j'ai créé aussi des fenêtres dans l'interface, comme la fenêtre qui vous permet de faire un rapprochement ou la fenêtre qui vous permet de choisir un rapprochement pour générer un rapport pdf (Cette dernière fonction nous n'avons pas pu terminer). J'ai fait des tests tout le temps à l'intérieur de l'application, testant le scénario d'utilisation et évitant certaines erreurs internes. J'ai créé la clôture des comptes à partir des fonctions de mes collègues et créez également des fonctions pour générer des rapports PDF des transactions.

- Création du diagramme UML
- Rapprochement des comptes (Interface)
- Clôture des comptes
- Génération des rapports PDF (À partir d'un compte ou tous les transactions)
- Lecture de fichier XML (Comptes / Transactions)
- Ecriture de fichier XML (Comptes / Transactions)
- Documentation des classes au niveau de l'interface
- Création de vidéo (Test)
- Solutions de différent types de bugs dans la version final

L'intensité du travail sur chaque tâche peut être vue dans le diagramme de bande ci-dessus

Oumnia Ait El Hihi:

Mes tâches consistaient en l'implémentation initiale des classes et méthodes de la partie Transaction...
La classe Transaction regroupait toutes les informations relatives à une transaction, notamment la date, la référence et le memo et les triplets (Compte, Débit, Crédit),
J'ai choisi de modéliser les triplets (Compte, Débit, Crédit) comme étant une nouvelle classe : Opération, afin de faciliter la vérification de la viabilité d'une transaction.
La classe Opération permettra donc d'effectuer les opérations, à travers les méthodes créditer et débiter une fois la transaction vérifiée (somme des crédits égale à la somme des débits).
La version initiale de la vérification du fonctionnement des transactions consistait à récupérer les id, les crédits et les débits des comptes concernés à raison de deux comptes au minimum par transaction et les stocker dans un vecteur d'Operations.
Ensuite viens la comparaison des débits/crédits, avant d'appeler tous les éléments du vecteur Operations avec la méthode effectuer opération ().
Dans le cas où la transaction n'était pas valide, un message d'erreur s'affichait et l'utilisateur devait donc ressaisir une transaction valide.

Mes versions initiales ont de l’être modifiées le long de la modification des autres parties du projet par les autres membres du groupe.

Annexes

Dans cette section, nous expliquerons un peu comment fonctionne chaque bouton du programme.

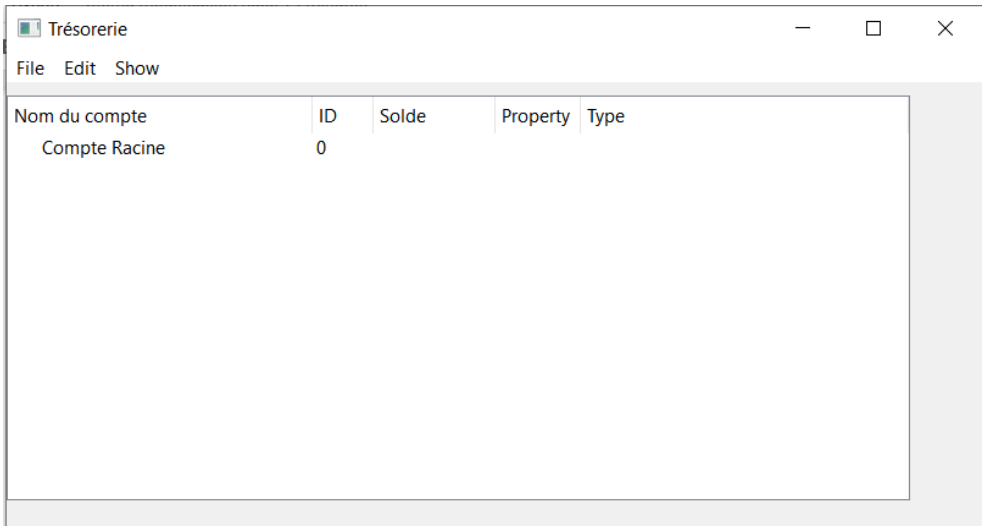


Image 2.1 Fenêtre principal

Dans file du menu, nous pouvons ouvrir une nouvelle fenêtre, enregistrer les modifications apportées, ouvrir un nouveau projet à partir de deux fichiers et générer des rapports PDF des transactions effectuées. Pour enregistrer vos modifications et ouvrir un nouveau fichier, v**ous commencez toujours par le fichier comptes.xml puis transactions.xml.** La fenêtre de sélection de fichiers en haut aura un titre indiquant le type de fichier attendu.

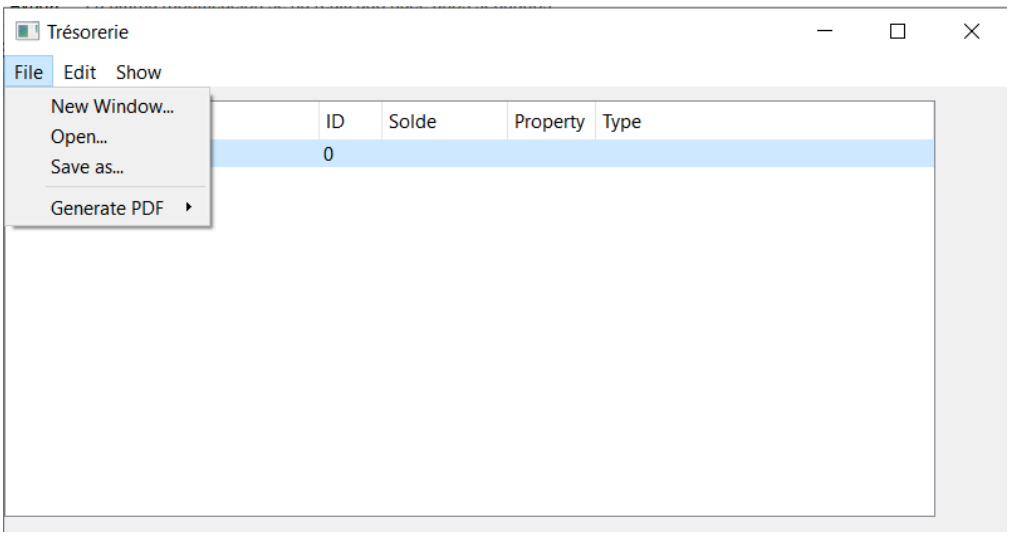


Image 2.2 Menu FIle

Dans la section “Edit”, nous pouvons ajouter des comptes, supprimer des comptes, ajouter une transaction ainsi que faire une clôture ou un rapprochement. Pour les deux premiers fonctions on doit cliquer sur une compte et après cliquer la fonction à réaliser.

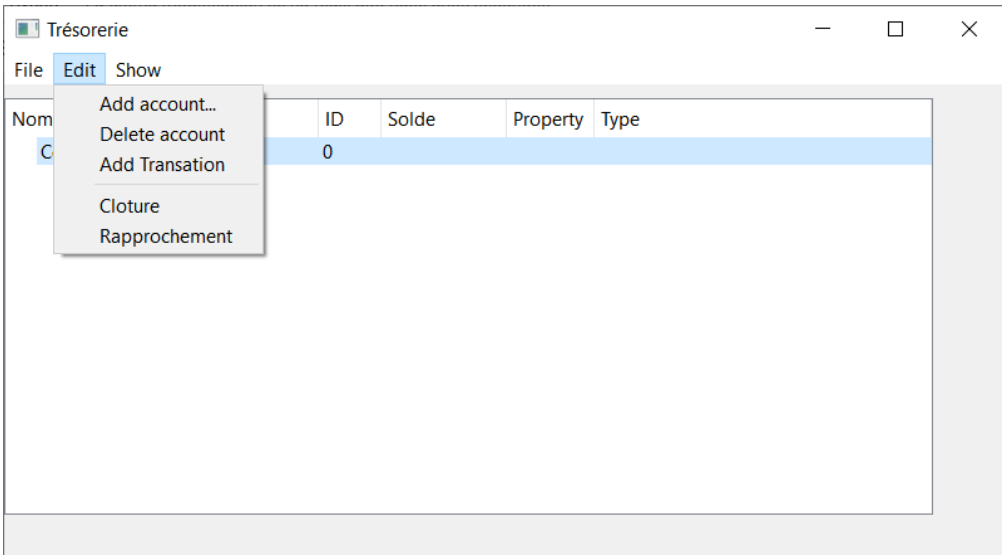


Image 2.3 Menu Edit

Dans la section “Show”, nous pouvons afficher la liste des transactions dans un QTableWidget et pouvoir les modifier.

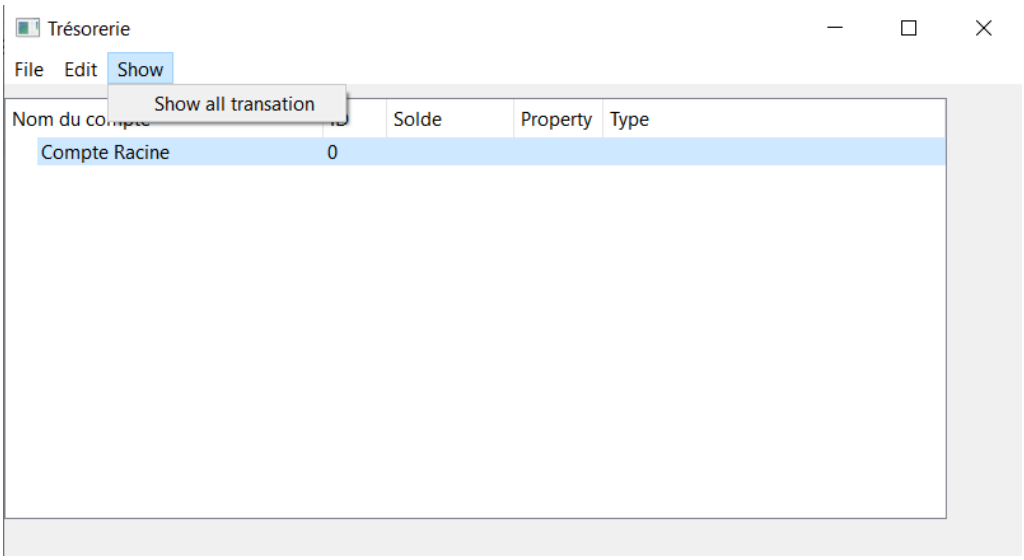


Image 2.4 Menu Show