LO21

Projet trésorerie

P20

Table des matières

Introduction **1**

1 Présentation de l'application **1**

1.1 Opérations implémentées 1

1.2 Opérations non implémentées 2

1.3 Pistes d'amélioration 2

2 Architecture du projet **2**

2.1 Architecture des comptes 3

2.1.1 La hiérarchie 3

2.1.2 Les postes 3

2.1.3 Le CompteManager 4

2.1.4 Les Mementos 4

2.1.5 Le HierarchieBuilder 5

2.1.6 Les Visiteurs 5

2.1.7 Les exceptions des Comptes 6

2.2 Architecture des transactions 6

2.2.1 Les transactions 6

2.2.2 Le TransactionManager 6

2.2.3 Les exceptions des Transactions 7

2.2.4 Les exceptions fichiers 7

2.3 Architecture de l'interface graphique 7

3 Planning **10**

Conclusion **10**

A Annexes **11**

A.1 Diagrammes 11

A.1.1 Diagrammes UML de l'architecture de l'application 11

A.1.2 Diagrammes UML de l'interface graphique 14

A.2 Présentation des livrables 17

Introduction

Ce document constitue le rapport de projet demandé dans le cadre de l'UV LO21, au printemps 2020. Nous y détaillons la conception d'un logiciel de trésorerie utilisant les principes de comptabilité en partie double. Dans un premier temps, nous présentons le produit fini avec ses capacités, ainsi que des pistes d'améliorations et des fonctionnalités qui n'ont pas été implémentées. Ensuite, nous décrivons l'architecture utilisée pour réaliser le logiciel, en nous intéressant plus particulièrement à la justification de cette architecture au regard de l'évolutivité de l'application. Enfin, nous livrons le planning observé de l'avancée du projet, ainsi qu'un résumé des contributions de chaque collaborateur à chaque étape.

1 Présentation de l'application

1.1 Opérations implémentées

Nous avons implémenté la création de comptes, réels ou virtuels (avec le bouton ajouter, depuis le menu et le raccourci Ctrl+N). Nous avons implémenté la possibilité de supprimer un compte (depuis le menu et avec le bouton supprimer) et de le déplacer dans la hiérarchie des comptes de même type : actif, passif, recette ou dépenses (avec le cliquer-déplacer ou par un menu dédié).

Nous avons implémenté les opérations liées au transactions. L'ajout (avec le bouton ajouter depuis le menu et avec le raccourci Ctrl+Shift+N), la suppression (avec le bouton supprimer et depuis le menu) et la modification des transactions (avec le bouton modifier, depuis le menu et le raccourcis Ctrl+M) sont possibles pour toutes les transactions qu'elles soient simples ou réparties.

Nous avons implémenté le calcul du solde d'un compte (correspondant à la dernière colonne des transactions de chaque compte), le rapprochement de compte (depuis le menu et le raccouri Ctrl+Alt+C) et la clôture de compte (depuis le menu et le raccourcis Ctrl+Alt+A).

Nous avons implémenté toutes les opérations comptables demandées c'est à dire l'affichage d'un relevé des opérations effectuées sur un compte, l'affichage du bilan d'un ensemble de comptes (actifs et passifs) et le calcul du résultat d'un ensemble de comptes (actifs, passifs, recettes et dépenses).

Nous avons implémenté la persistance des informations avec la sauvegarde de la hiérarchie des comptes et des transactions associées en XML. Cette sauvegarde se fait dans un fichier choisi par l'utilisateur et qui peut être chargé au lancement de l'application pour restaurer un ensemble de comptes préalablement établis. Le fichier de sauvegarde est préférentiellement de type *.business* ou *.xml* mais peut être d'un type différent tant que le contenu est identique.

Nous avons implémenté la possibilité de choisir la devise utilisée pour les transactions (dans les paramètres). Les devises possibles sont l'euro EUR, le dollar américain USD, le franc Suisse CHF, la livre sterling GPB, le yen japonais JPY, le franc Français FRF, le bitcoin BTC et le simsflouz §. Il est également possible de paramétrer le format de date pour les affichages au

sein de l'application.

Nous avons implémenté la sauvegarde du contexte de l'application, notamment le choix de la devise et du format de date utilisé qui est enregistré dans le fichier de sauvegarde en XML et qui peut être chargé au démarrage de l'application, ou à n'importe quelle moment.

Nous avons implémenté l'édition de documents de comptabilité (les relevés de comptes et les bilans, au format PDF), ainsi que l'affichage des données de comptabilité qui est possible depuis le menu ou avec les raccourcis indiqués précédemment.

1.2 Opérations non implémentées

Par soucis de cohérence nous avons empêché la saisie de transactions à des dates futures, la demande de création d'une transaction à une date supérieure à la date du jour déclenche une exception et empêche la création de la transaction. Il nous est donc impossible de faire certaines des opérations demandées dans le scénario d'utilisation du projet (par exemple nous ne pouvons pas saisir le versement d'une subvention municipale en décembre 2020). Nous avons donc décidé de saisir toutes ces transactions avec l'année 2019 au lieu de 2020. Un fichier correspondant à la sauvegarde de la hiérarchie des comptes et des transactions du scénario d'utilisation de l'application se trouve dans la partie *Backup* de notre projet.

1.3 Pistes d'amélioration

Tout d'abord, nous pourrions améliorer la portabilité de l'application, qui fonctionne très bien sous Windows mais a quelques problèmes sous macOS. De plus, la fonctionnalité de *Drag&Drop* que nous souhaitions mettre en place pour le déplacement des comptes est parfois capricieuse et pourrait être remaniée. Nous pourrions également générer les documents comptables au format PDF détaillant la hiérarchie des comptes, ou le résultat d'un compte. Nous avons laissé la possibilité d'ajouter d'autres devises, et afin de rendre l'application accessible à un plus large public, il pourrait être intéressant de traduire les différents messages affichés. Par ailleurs, un bouton d'aide est affiché par défaut sur les fenêtres de l'application, mais aucun comportement n'a été défini. Nous pourrions implémenter des modules d'aides, accessibles en cliquant sur ce bouton. Nous pourrions autoriser la création de transactions dans le futur, qui seraient différenciées des autres par leur affichage pour permettre à l'utilisateur de faire facilement des budgets prévisionnels et/ou de prendre en compte des dépenses contraintes (dépenses qui sont obligatoirement faites, comme les frais de personnel ou les frais bancaires).

2 Architecture du projet

Notre architecture était initialement divisée en deux : une partie centrée sur la gestion des comptes, et une autre sur les transactions. Ensuite, nous y avons ajouté une partie dédiée à l'aspect graphique de l'application.

2.1 Architecture des comptes

2.1.1 La hiérarchie

D'après les exigences du sujet, les comptes doivent pouvoir être organisés en arborescence, avec des comptes virtuels (les noeuds), et des comptes réels (les feuilles). Dans l'application, nous voulons donc être capables de manipuler cette arborescence, sans savoir précisément quel type de compte (réel ou virtuel) nous manipulons.

Pour répondre à ces exigences, nous utilisons le design pattern *Composite*, les comptes virtuels étant les objets composés, et les compte réels les composants. Les deux types de comptes héritent d'une classe mère abstraite CompteHierarchie, qui désigne un élément quelconque de cette hiérarchie. Elle définit une interface de manipulation de ses classes filles (CompteReel et CompteVirtuel), en définissant quelques méthodes communes simples, et en déclarant des méthodes virtuelles pures. Ensuite, on définit dans CompteReel les méthodes permettant la gestion de solde, et on interdit les méthodes permettant la gestion des fils en jetant des exceptions appropriés. Dans CompteVirtuel, on fait l'inverse.

Cette organisation nous permet de gérer très facilement une hiérarchie d'objets imbriqués les uns dans les autres. Nous pourrions imaginer définir une autre classe héritant de CompteHiérarchie, et il serait alors possible de l'intégrer dans les fils des comptes virtuels avec peu de modifications.

Voir la figure 2 et la documentation pour plus de détails.

2.1.2 Les postes

Nous souhaitons être en mesure de regrouper les comptes selon quatre catégories : Actif, Passif, Dépense et Recette. Chacune de ces catégories définit le comportement à adopter pour le compte concerné en cas de débit et de crédit, ainsi que des contraintes sur la hiérarchie (une compte virtuel ne peut regrouper que des comptes de même type que lui, sauf la racine). Cependant, nous voulons garder ce comportement séparé de l'organisation hiérarchique pour conserver une certaine souplesse : il est hors de question de définir 4 sous-classes de comptes réels et 4 sous-classes de comptes virtuels, car cela rendrait le système beaucoup plus rigide.

Pour répondre à ce besoin, nous utilisons le design pattern *Bridge*, qui nous permet de découpler le comportement d'un compte dépendant de son poste et son comportement réel ou virtuel dans la hiérarchie. Ainsi, chaque compte de la hiérarchie est associé à un poste particulier. Nous avons regroupé les postes d'actifs et de dépenses car leur comportement au regard du débit et du crédit est identique. De même pour les postes de passifs et de recettes. La racine est un poste un peu particulier, qui ne définit pas d'opération de crédit ou de débit, et permet au compte concerné d'avoir n'importe quel poste parmi ses fils. Toutes ces classes sont des *Singletons* : elles définissent seulement un comportement sans posséder d'attributs et nous n'avons donc aucune raison de les instancier plusieurs fois.

Cette structure nous permet de changer facilement le comportement de tous les comptes de même poste, ou éventuellement d'en ajouter de nouveaux, indépendamment du reste de l'architecture.

Voir la figure 3 et la documentation pour plus de détails.

2.1.3 Le CompteManager

La hiérarchie des comptes est relativement complexe à manipuler, et certaines opérations sont interdites ou ne devraient pas être réalisées seules. Nous ne pouvons donc pas laisser cette hiérarchie sous la responsabilité seule de l'utilisateur.

Le CompteManager est un *Singleton*, dont le rôle est de fournir une interface pour la gestion de cette hiérarchie. C'est par lui que doit passer l'utilisateur pour récupérer un pointeur sur un compte, afficher ou modifier la hiérarchie, la charger ou sauvegarder, effectuer les opérations comptable complexes etc...Etant donné que le père d'un compte peut changer pendant l'exécution de l'application, c'est lui qui a la responsabilité de gérer leur durée de vie. Plus tard, ce sont ses méthodes que l'interface graphique utilisera selon les actions de l'utilisateur.

La présence de ce manager nous permet d'ajouter facilement des traitements sur notre hiérarchie. Par exemple, si un nouveau document de comptabilité est nécessaire, il suffit de lui ajouter une méthode définissant le traitement à effectuer (qui pourra tirer parti des opérations de base déjà implémentées comme la récupération d'un compte correspondant à ID donné).

Voir la figure 2 et la documentation pour plus de détails.

2.1.4 Les Mementos

Nous avions besoin de pouvoir gérer le solde des comptes réels dans le temps, en fonction des transactions effectuées. Cependant, quelques contraintes s'ajoutaient à cela : nous ne voulons pas recalculer le solde d'un compte à partir de 0 à chaque fois car c'est une opération que nous serons amenés à réaliser souvent par la suite, nous voulons être capable d'afficher le solde après chaque transaction, et nous savons que lors d'un rapprochement on fige le solde du compte avant une certaine date.

Nous choisissons donc d'utiliser des *Mementos*. Les comptes réels disposent d'une liste de Mementos pour gérer l'évolution de leur solde dans le temps. Dans cette liste, le dernier Memento contient une estimation dynamique du solde au moment actuel, alors que les précédents contiennent des soldes figés par des rapprochements (ou 0 à la création du compte). Ainsi, lorsque nous souhaitons accéder au solde actuel d'un compte, nous pouvons simplement lire la valeur du solde du dernier Memento. Lorsque nous souhaitons afficher le solde à un moment passé, nous trouvons le Memento antérieur le plus proche dans la liste, et nous calculons le solde à partir de ce point. Afin de gérer ce système, nous avons également besoin d'un Originator utilisé par le CompteManager, et qui permet d'effectuer les opérations sur cette liste de Mementos.

L'inconvénient de cette méthode est que nous devons mettre à jour le solde actuel lorsque nous effectuons ou modifions une transaction. Cependant, elle s'avère bien plus simple d'utilisation dans le code par la suite. Nous pourrions facilement imaginer modifier ce Memento pour y stocker plus d'informations si nécessaire.

Voir la figure 2 et la documentation pour plus de détails.

2.1.5 Le HierarchieBuilder

Nous avons choisi de représenter la hiérarchie des comptes comme un arbre. Cependant, la construction de cet arbre peut s'avérer compliqué : elle s'étale sur plusieurs niveaux d'imbrication, et doit respecter des règles précises.

Afin de ne pas laisser cette responsabilité directement au CompteManager, nous avons décidé de créer une classe HierarchieBuilder qui regrouperait ces méthodes. Elle nous permet de créer un arbre vide, ou à partir d'un fichier. Elle dispose également d'une méthode permettant de créer un compte de type et de poste donné, dont elle se sert elle-même pour créer la hiérarchie à partir d'un fichier. En ce sens, son fonctionnement est proche de celui d'un *Builder* : elle commence par créer une racine, puis ajoute des éléments fils en utilisant une de ses méthodes, avant de renvoyer l'arbre final.

Le principal avantage de cette approche est qu'elle nous permet d'intervenir facilement sur le processus de création de l'arbre. Nous pourrions par exemple imaginer lui ajouter des méthodes permettant de le créer à partir d'un autre type de fichier. Le CompteManager ne sait pas comment est créée cette hiérarchie, et ces changements seraient donc totalement transparents pour lui.

Voir la figure 2 et la documentation pour plus de détails.

2.1.6 Les Visiteurs

L'organisation en arbre nécessite un traitement spécial pour accéder à chaque élément. En effet, le parcours d'un élément nécessite d'être en mesure de parcourir ses fils (eux mêmes pouvant avoir des fils), tout en étant capable de retourner à l'élément père ensuite.

Nous utilisons pour cela une architecture sur le modèle *Visitor*. Les comptes réels et virtuels disposent d'une méthode pour accepter un visiteur en lui passant leur adresse. Le visiteur dispose de deux méthodes pour visiter un compte, en fonction de son type. Il ne reste alors qu'à définir un comportement pour les types réels (souvent une action simple), et pour les types virtuels (la visite de tout ses fils par exemple). Il est également possible d'ajouter des attributs à ces visiteurs, dont on pourra récupérer les valeurs suite à la visite. Nous avons défini 6 visiteurs en fonction de nos besoins : un visiteur de libération de mémoire, un visiteur d'affichage, un visiteur qui recherche l'adresse d'un compte d'ID passé en paramètre, un visiteur qui calcule le solde d'un compte, un visiteur qui renvoie le père d'un compte, et enfin un visiteur qui construit une liste de tous les comptes d'un poste donné.

Nous nous sommes plusieurs fois étonné de la facilité d'utilisation de ce design pattern. En effet, une fois la classe mère et les méthodes d'acceptation définies dans CompteHierarchie, il suffit de quelques lignes pour implémenter un nouveau comportement. Par exemple, nous avons choisi d'effectuer la sauvegarde dans le CompteManager, mais nous aurions pu appeler un Visiteur qui aurait parcouru la hiérarchie en écrivant dans le fichier.

Voir la figure 4 et la documentation pour plus de détails.

2.1.7 Les exceptions des Comptes

Dans le but de savoir quels sont les problèmes rencontrés par les différentes fonctions, nous avons défini une classe mère Exception, qui permet de stocker un code d'exception et un message, et de les récupérer dans un *catch*.

Du côté des comptes, deux classes héritent de cette classe mère : ExceptionHiérarchie et ExceptionComptabilité. ExceptionHierarchie est utilisée lorsqu'une exception concerne la hiérarchie des comptes, et les codes de 0 à 19 lui sont réservés. On l'utilise par exemple lorsqu'on ne respecte pas les règles de création d'un compte, lorsqu'on ne trouve pas un compte dans la hiérarchie, qu'on tente d'accéder aux fils d'un compte réel, etc...ExceptionComptabilité est utilisée lorsqu'une exception concerne les opérations comptable, et les codes de 20 à 39 lui sont reservés. On peut l'utiliser si on tente d'accéder au solde d'un compte virtuel, si on veut modifier le solde de la racine, ou si la date d'un relevé est incorrecte...

Cette organisation nous permet de facilement ajouter des exceptions lorsqu'une méthode repère un cas critique, et de les traiter par la suite en fonction du problème relevé.

Voir la figure 1 et la documentation pour plus de détails.

2.2 Architecture des transactions

2.2.1 Les transactions

Notre application est destinée à gérer des opérations de comptabilité en partie double, la gestion des transactions est donc un point vital. Nous savons qu'il doit être possible de réaliser des transactions simples (entre deux comptes), ou réparties (avec plusieurs comptes émetteurs et récepteurs), tout en respectant un certain nombre de contraintes sur les débits et crédits.

Pour faire cela, nous avons décidé qu'une transaction doit être composée de multiples opérations (au moins 2, une de débit et une de crédit). Une opération concerne un compte, ainsi qu'un débit ou un crédit qui sont automatiquement rééquilibrés en cas de saisie dans les deux champs en même temps. Ensuite, une transaction comporte une liste de ces opérations, et d'autres informations comme une référence et un mémo explicatif. Nous stockons également sa date avec une structure `time_t`, et il est donc impossible d'enregistrer des transactions avant le 1er janvier 1970. Nous avons également fait le choix de refuser les dates dans le futur.

Voir la figure 5 et la documentation pour plus de détails.

2.2.2 Le TransactionManager

Comme pour les comptes, nous ne pouvons pas laisser ces transactions directement aux mains de l'utilisateur. En effet, il en faut pas qu'il soit capable d'en supprimer ou d'en ajouter sans mettre à jour le solde des comptes concernés, et il y a également des contraintes sur la modification. Chaque transaction est unique, il est impossible de créer deux transactions avec les mêmes informations, ce qui permet d'éviter les erreurs et la fraude.

Nous utilisons donc ici aussi un objet manager, qui est un *Singleton*. Il contient une liste de

transactions, et est responsable de leur cycle de vie (création, modification, destruction). Il a aussi pour rôle de construire différentes listes pour itérer sur ces transactions. Par exemple, il permet de retrouver toutes les transactions qui concernent un compte donné. Tout comme le CompteManager, il a aussi recours aux services d'un *Builder*, le TransactionBuilder, pour instancier les transactions ou charger la liste depuis un fichier de sauvegarde.

Cette organisation a les mêmes avantages que les CompteManager : il est très simple d'ajouter des méthodes au manager en fonction des besoins, ce qui permet une certaine souplesse d'utilisation. En cas de besoin d'un nouveau type de parcours par exemple, il suffit d'ajouter une méthode construisant cette liste.

Voir la figure 5 et la documentation pour plus de détails.

2.2.3 Les exceptions des Transactions

Comme les exceptions des comptes, les exceptions côté transactions (ExceptionTransaction) héritent de la classe Exception. Les codes réservés sont numérotés de 40 à 59. Elles sont utilisées lorsqu'une action pose un problème, par exemple qu'une allocation de mémoire échoue, qu'on tente de créer une transaction dans le futur, qu'une recherche échoue. . .Le premier code sert à signaler une exception qui ne correspond pas à un code existant.

Nous pourrions imaginer d'autres exceptions qui au lieu d'être considérées comme des exceptions inconnues, seraient considérées comme un nouveau type d'exception, avec un autre traitement pour faciliter l'utilisation de l'application pour un utilisateur inexpérimenté. Cependant les codes d'exception actuels nous permettent déjà d'effectuer les vérifications nécessaires.

Voir la figure 1 et la documentation pour plus de détails.

2.2.4 Les exceptions fichiers

Comme les exceptions précédentes, les ExceptionFichier héritent de la classe mère Exception. Les codes correspondant vont de 60 à 79. Ces exceptions sont utilisées lorsqu'il y a un problème avec la lecture ou l'écriture dans un fichier de sauvegarde. Elles sont par exemple lancées lorsqu'un fichier est introuvable ou inaccessible, ou que la syntaxe du fichier lu ne correspond pas à celle attendue. Elles sont surtout utilisées par le CompteManager et le TransactionManager lors du chargement et de la sauvegarde.

On notera au passage que dans la couche graphique toutes les exceptions compréhensibles par l'utilisateur sont affichées dans des fenêtres de *warning*, afin de rendre compréhensible le comportement de l'application lors d'un échec.

Voir la figure 1 et la documentation pour plus de détails.

2.3 Architecture de l'interface graphique

Nous avons choisi d'organiser notre application autour d'une vue principale, présentant la hiérarchie des comptes ainsi que les transactions qui y sont liés. Des boutons permettant d'accéder

rapidement aux fonctionnalités d'ajout/suppression des comptes sont également présents sur cette fenêtre pour permettre d'utiliser facilement les fonctionnalités principales. Les fonctions plus complexes sont accessibles via les menus déroulants. La plupart de ces actions déclenchent l'apparition d'une nouvelle fenêtre permettant de saisir les informations nécessaires.

Nous avons donc une classe principale (MainWindow), qui est en interaction avec le CompteManager et le TransactionManager. Cette fenêtre permet aussi de gérer l'apparition des sous-fenêtres, et est donc responsable de leur cycle de vie. On peut répartir ces sous-fenêres en plusieurs catégories comme suit :

La gestion des comptes : Les opérations de gestion des comptes sont accessibles depuis l'onglet **Compte** du menu déroulant, sans avoir à sélectionner les comptes concernés. Il est possible de créer un compte, en précisant dans quel dossier le compte devra être inclus, le nom du compte, le type du compte (virtuel : dossier, réel : compte) et le poste du compte. Lors de la création d'un compte de type actif ou passif, l'application propose à l'utilisateur d'initialiser ce compte.

Il est également possible de supprimer un compte s'il ne contient pas de transactions. Avec la couche graphique, la suppression n'est pas automatique, une boite de dialogue demande à l'utilisateur de valider son choix. La création et la suppression de compte sont également accessibles depuis la fenêtre principale, en sélectionnant le compte à supprimer ou le dossier parent dans la hiérarchie.

De plus, il est possible de déplacer un compte dans la hiérarchie des comptes à condition que les postes des comptes impliqués dans le déplacement soient compatibles. La couche graphique permet d'informer l'utilisateur de la réussite du déplacement.

Voir la figure 10 et la documentation pour plus de détails.

La gestion des transactions : Les opérations de gestion des transactions sont accessibles depuis l'onglet **Transaction**, sans avoir à sélectionner les transactions concernées. Il est possible d'ajouter une transaction en renseignant une référence, un memo explicatif ainsi que des montants de débit et de crédit cohérents. La couche graphique renseigne par défaut la date et l'heure avec la date de la création de la transaction. De plus, le premier compte de la hiérarchie est sélectionné par défaut, pour toutes les opérations d'une transaction.

La suppression de transaction est possible sans condition, quelle que soit la transaction. Cependant, la couche graphique, demande à l'utilisateur de valider son choix, via une boite de dialogue.

Enfin la modification de transaction est possible. Cette modification peut être partielle (un ou plusieurs attributs) ou totale (tous les attributs) à condition que la transaction modifiée ne soit pas identique à une transaction pré-existante. L'ajout, la suppression et la modification de transaction sont également possibles depuis la fenêtre principale en sélectionnant la transaction à modifier ou à supprimer dans la liste des transactions.

Voir la figure 12 et la documentation pour plus de détails.

La gestion comptable : Les opérations de gestion comptable sont accessibles depuis l'onglet **Document**. Il est possible de rapprocher les transactions d'un compte et de les modifier si cela est nécessaire. Cependant, un rapprochement ne peut s'effectuer que sur des transactions qui n'ont pas déjà été rapprochées et se fait compte par compte. En revanche,

le rapprochement s'effectue sur toutes les transactions associées à un compte : il n'est pas possible d'en rapprocher certaines et pas d'autres. De plus, un rapprochement empêche la modification ou la suppression ultérieure des transactions. La couche graphique permet de demander à l'utilisateur de valider son choix, avec une boite de dialogue. Enfin, il est également possible de clôturer la hiérarchie des comptes, c'est à dire de rassembler toutes les transactions et de placer le montant correspondant dans un compte d'excédent ou de déficit. Seules les transactions d'initialisation des comptes ne sont pas incluses dans la clôture.

Voir la figure 9 et la documentation pour plus de détails.

L'édition de documents comptables : Les opérations d'édition de documents comptables sont accessibles depuis l'onglet **Document**. Il est possible de faire un bilan de la hiérarchie des comptes c'est à dire d'afficher et de sommer les soldes de tous les comptes d'actifs et de passifs. Un bilan peut être effectué à la date du jour ou à une date antérieure. Il est possible d'exporter les informations relatives à un bilan sous la forme d'un document PDF. Il est également possible de faire des relevés de comptes, c'est à dire d'afficher les soldes de tous les comptes de recettes et de dépenses sur une période choisie par l'utilisateur. Il est possible d'exporter les informations relatives à un relevé sous la forme d'un document PDF. De plus, il est possible de calculer le résultat de la hiérarchie de comptes, c'est à dire de sommer les soldes de tous les comptes en fonction de leur poste (actif, passif...) et d'afficher le résultat de cette somme sous la forme d'un excédant ou d'un déficit. Cette opération n'a aucune incidence sur les transactions ou sur les comptes.

Voir la figure 8 et la documentation pour plus de détails.

La gestion de l'application : La gestion de l'application est possible depuis l'onglet **Fichier**. Il est possible de sauvegarder la hiérarchie des comptes, les transactions saisies et les paramètres choisis par l'utilisateur. Cette sauvegarde peut être faite dans n'importe quel dossier choisi par l'utilisateur, dans n'importe quel fichier qui supporte le format de sauvegarde.

L'application étant centrée autour d'une fenêtre principale (MainWindow) et de plusieurs sous fenêtres dépendantes de la fenêtre principale, il n'était pas nécessaire de sauvegarder la fenêtre courante lors de la fermeture de l'application, afin de pouvoir la restaurer à la réouverture. En effet, un utilisateur qui fermerait l'application serait forcément sur la fenêtre principale, qui est ouverte, par défaut, au démarrage de l'application. La sauvegarde est faisable à tout moment.

Il est également possible de modifier les paramètres de l'application c'est à dire la devise utilisée parmi les devises disponibles (EUR, USD...) et le format de la date parmi trois formats possibles. La couche graphique permet de définir la devise par défaut avec l'euro et le format de date par défaut avec le format standard utilisé en France. Le choix des paramètres est automatiquement sauvegardé avec les données comptables. De plus, il est possible de charger une sauvegarde au démarrage de l'application ou de changer de sauvegarde à n'importe quel moment.

Voir la figure 11 et la documentation pour plus de détails.

3 Planning

UML : Première version de l'UML : 16 avril 2020. Actualisation permanente depuis.

Partie console — comptes et transactions : Premiers fichiers : 25 avril 2020. Actualisation permanente depuis.

Documentation : Définition de la syntaxe : 25 avril 2020. Premiers fichiers : 26 avril 2020. Actualisation permanente depuis.

Partie graphique — vues et adaptation du code : Premiers fichiers : 12 mai 2020. Actualisation permanente depuis.

Notre projet a commencé par un questionnement autour de l'architecture à adopter : il fallait qu'elle permette de répondre aux exigences du sujet, tout en restant relativement souple et puisse évoluer en fonction de nos avancées. Lors des premières réunions, nous avons pesé le pour et le contre de différentes représentations UML, menant à l'élaboration de l'architecture présentée précédemment. Un rapide prototype a également été élaboré pour vérifier la cohérence de ces choix. Il a servi de base à la portion du code concernant la gestion des comptes.

Nous avons ensuite implémenté cette architecture de façon à avoir une première version console qui soit fonctionnelle et nous permette de tester le coeur de l'application. Nous l'avons pour cela découpé en deux parties, une partie Compte et une partie Transaction, et avons formé deux groupes pour travailler en parallèle. Deux semaines plus tard, nous sommes entré en phase de test pour l'application console. Nous avons identifié de nombreuses situations, et le comportement attendu en réponse, avant de les tester systématiquement et noter le résultat. Les bugs identifiés à cette étape ont été corrigé sur le coup.

Enfin, nous avons ajouté Qt à notre projet. Les vues ont été réalisées avec Qt Designer, puis connectées aux fonctionnalités développées précédemment. Certaines ont dû être modifiées pour fonctionner avec une interface graphique, ou pour tirer parti des fonctionnalités déjà implémentées dans Qt. De nombreux tests et correction de bugs ont été réalisés pendant cette période. En parallèle de cela, étant sûrs que l'architecture principale de l'application ne changerait plus, nous avons commencé à rédiger le présent rapport.

Conclusion

Ce projet nous a permis de progresser en C++, de nous familiariser avec QtCreator et de découvrir QtDesigner. Nous avons également appris à travailler en groupe sur un projet long et conséquent, en nous organisant à distance. Nous avons su nous répartir le travail à faire et respecter un cahier des charges précis. Nous avons dû nous organiser en fonction d'un planning imposé, et avons découvert les avantages et les inconvénients de certains outils (plantUML, Git, VLC...). Enfin, nous avons dû travailler sans jamais nous voir, en étant tous à des endroits différents.

A Annexes

A.1 Diagrammes

A.1.1 Diagrammes UML de l'architecture de l'application

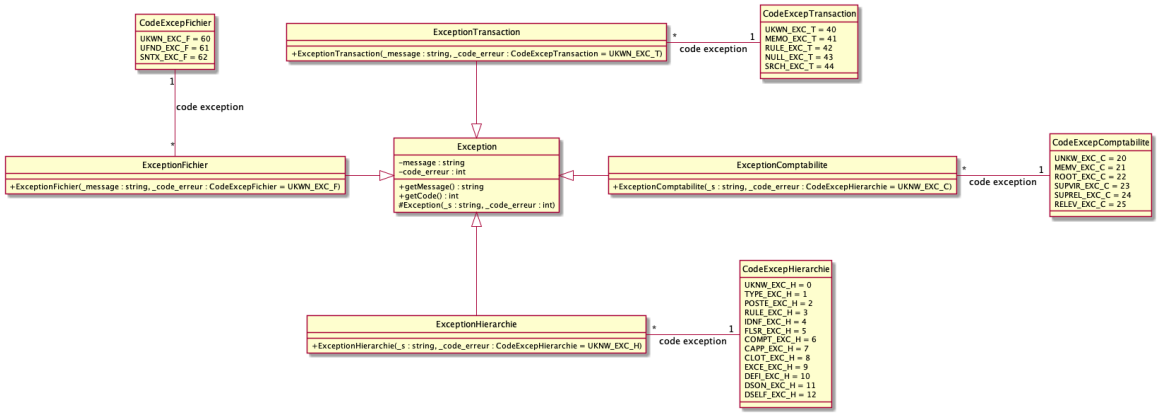

FIGURE 1 – Diagramme UML des exceptions

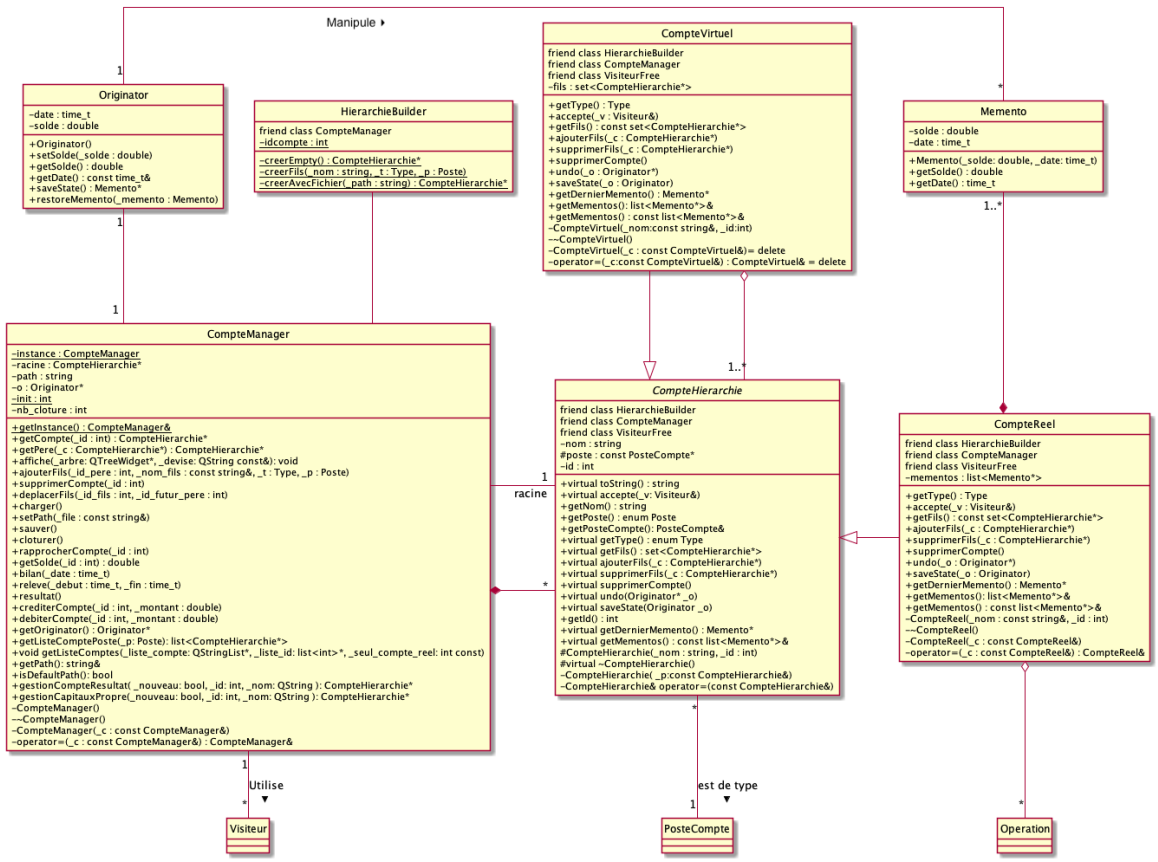

FIGURE 2 – Diagramme UML de la hiérarchie des comptes

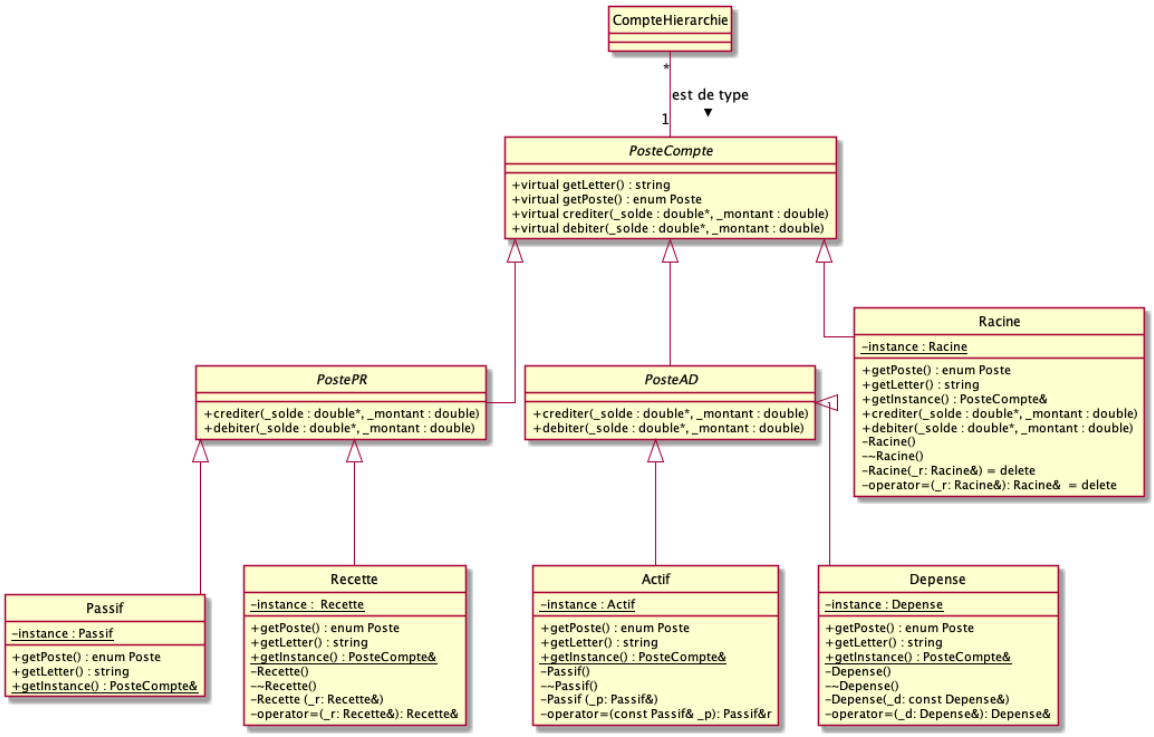

FIGURE 3 – Diagramme UML des postes

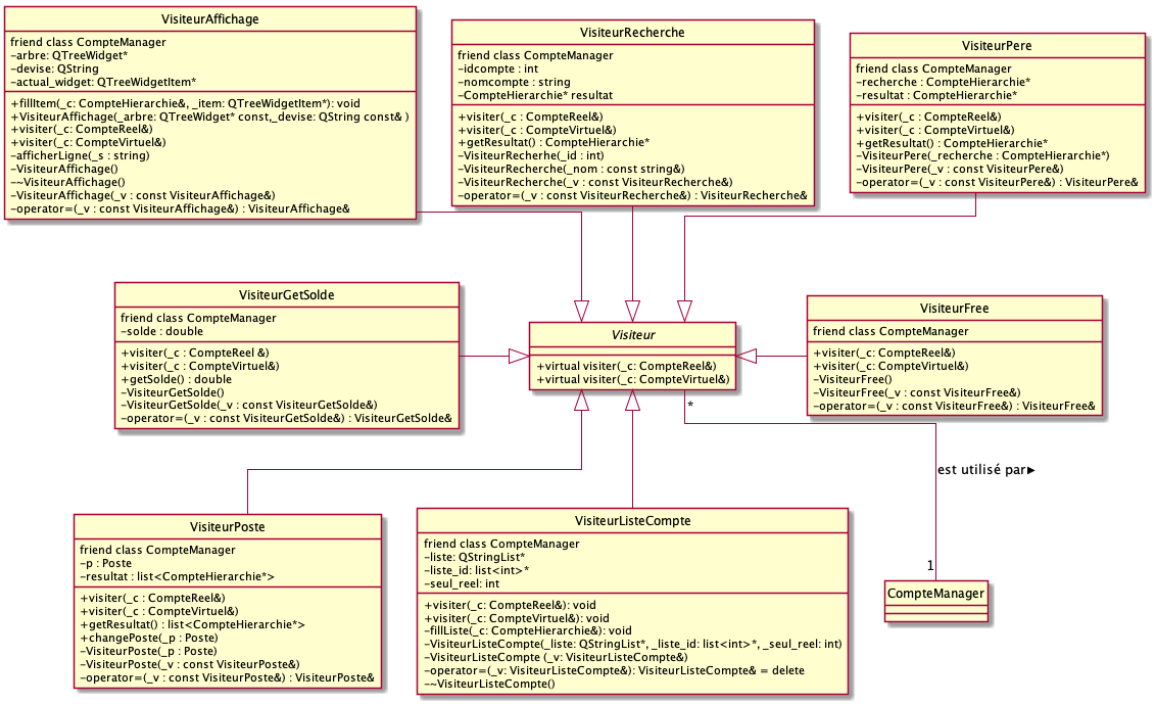

FIGURE 4 – Diagramme UML des visiteurs

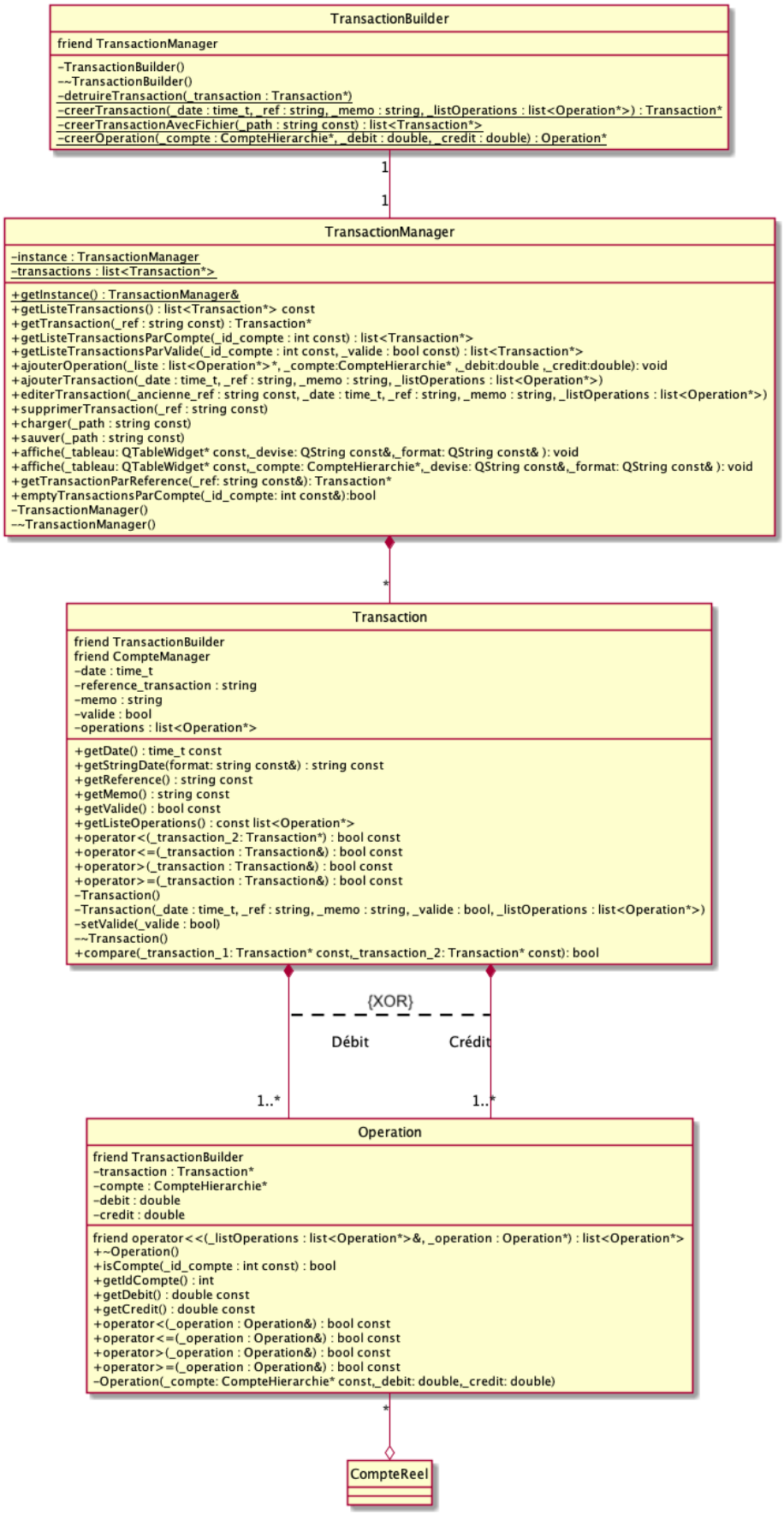

FIGURE 5 – Diagramme UML des transactions

A.1.2 Diagrammes UML de l'interface graphique

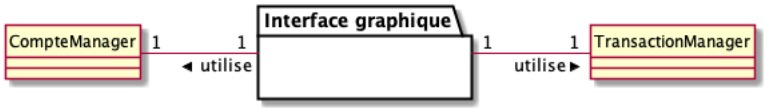

FIGURE 6 – Interactions entre l'application et les Managers

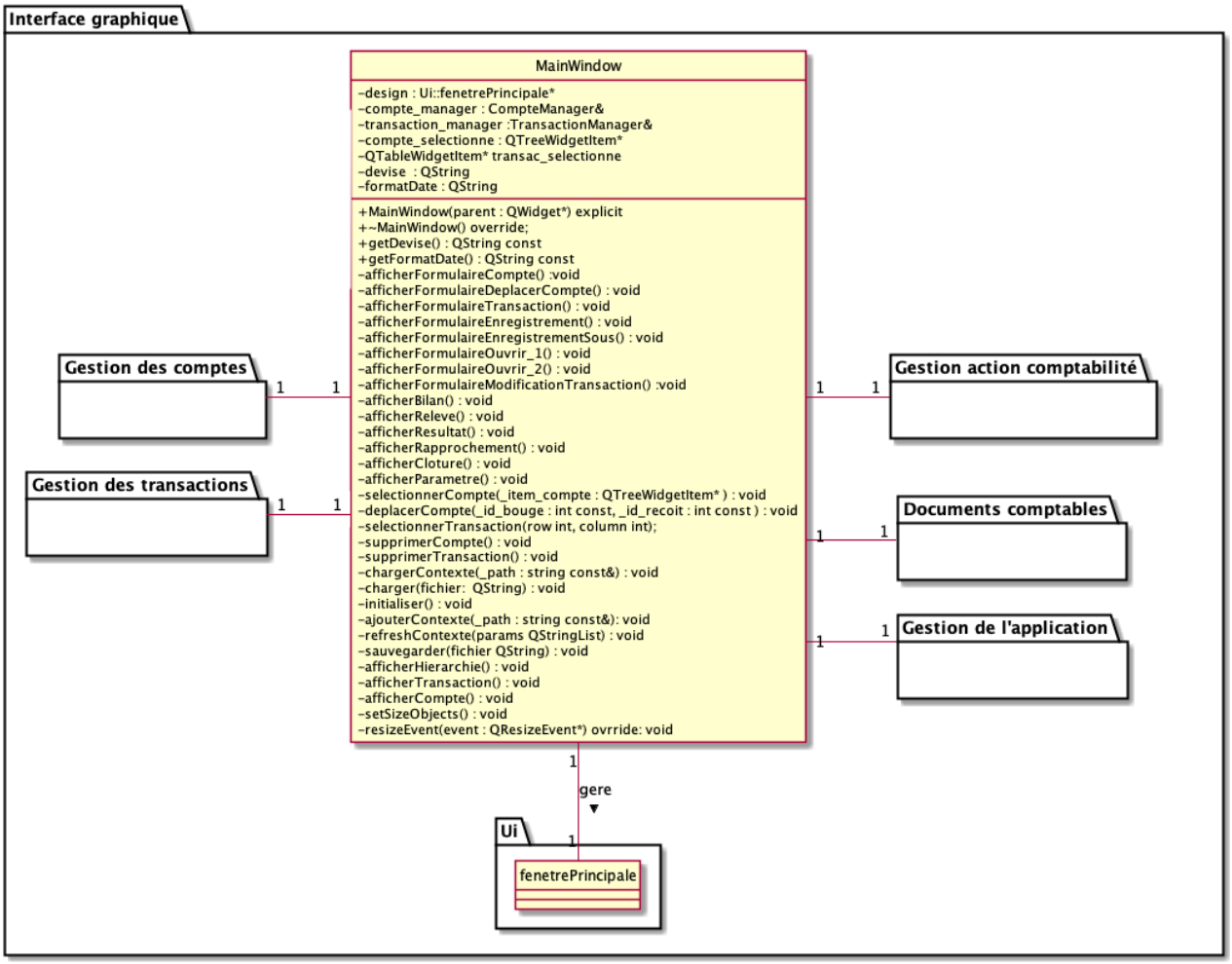

FIGURE 7 – Interface graphique

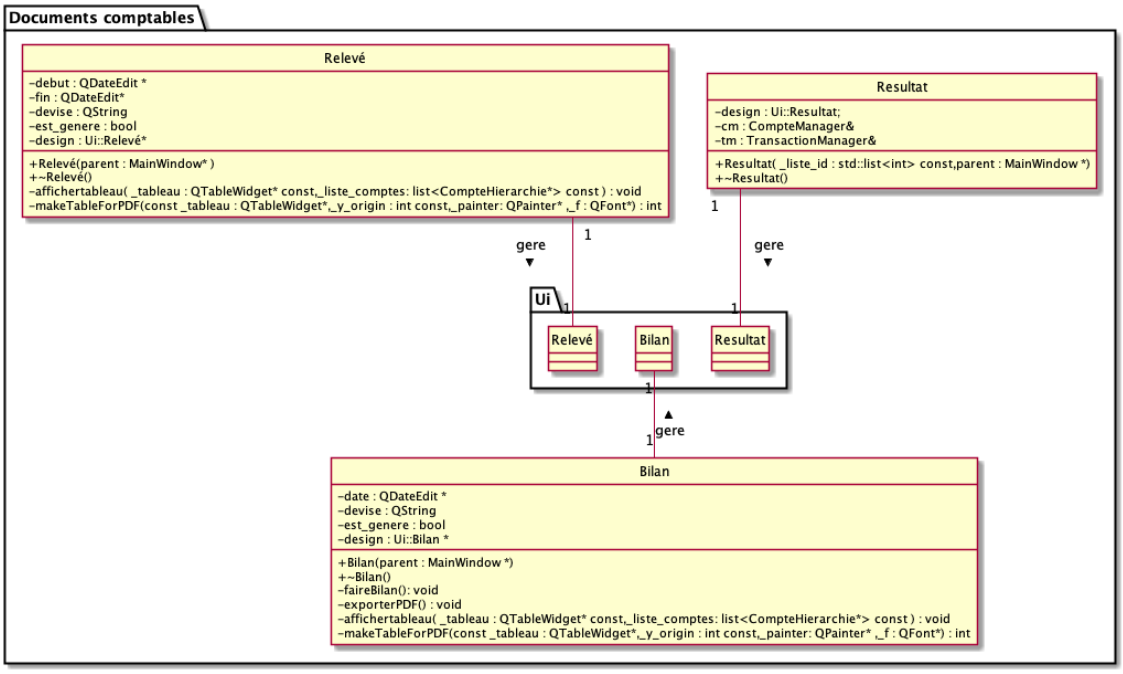

FIGURE 8 – Gestion des documents

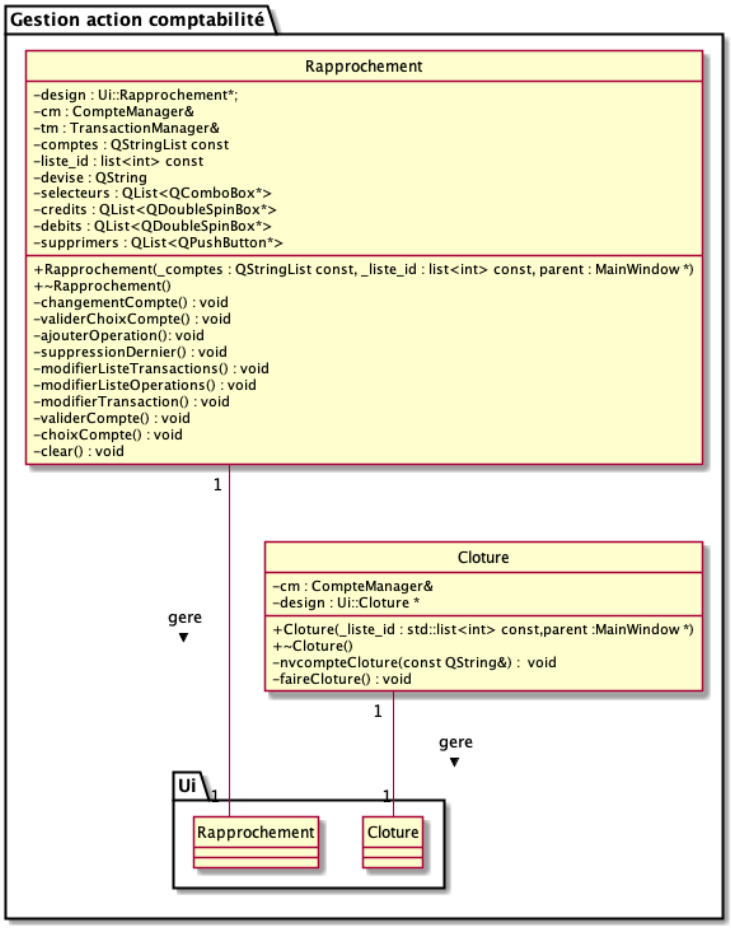

FIGURE 9 – Gestion des opérations de comptabilité

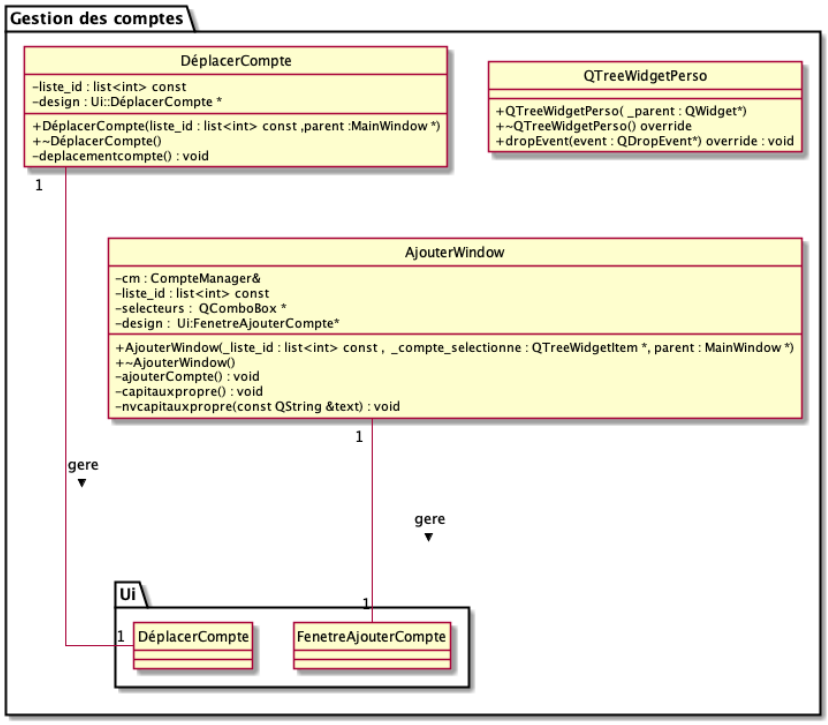

FIGURE 10 – Gestion des comptes

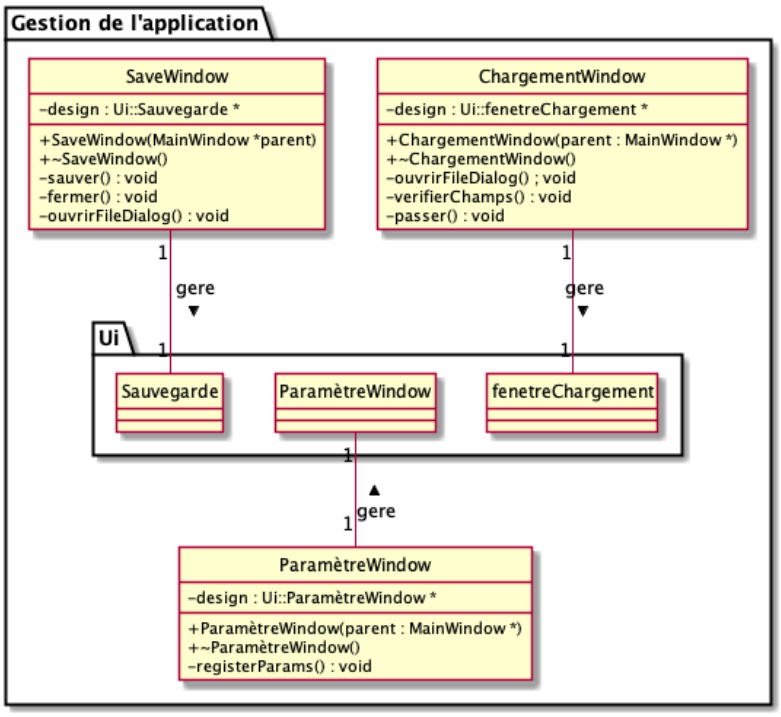

FIGURE 11 – Gestion des fichiers

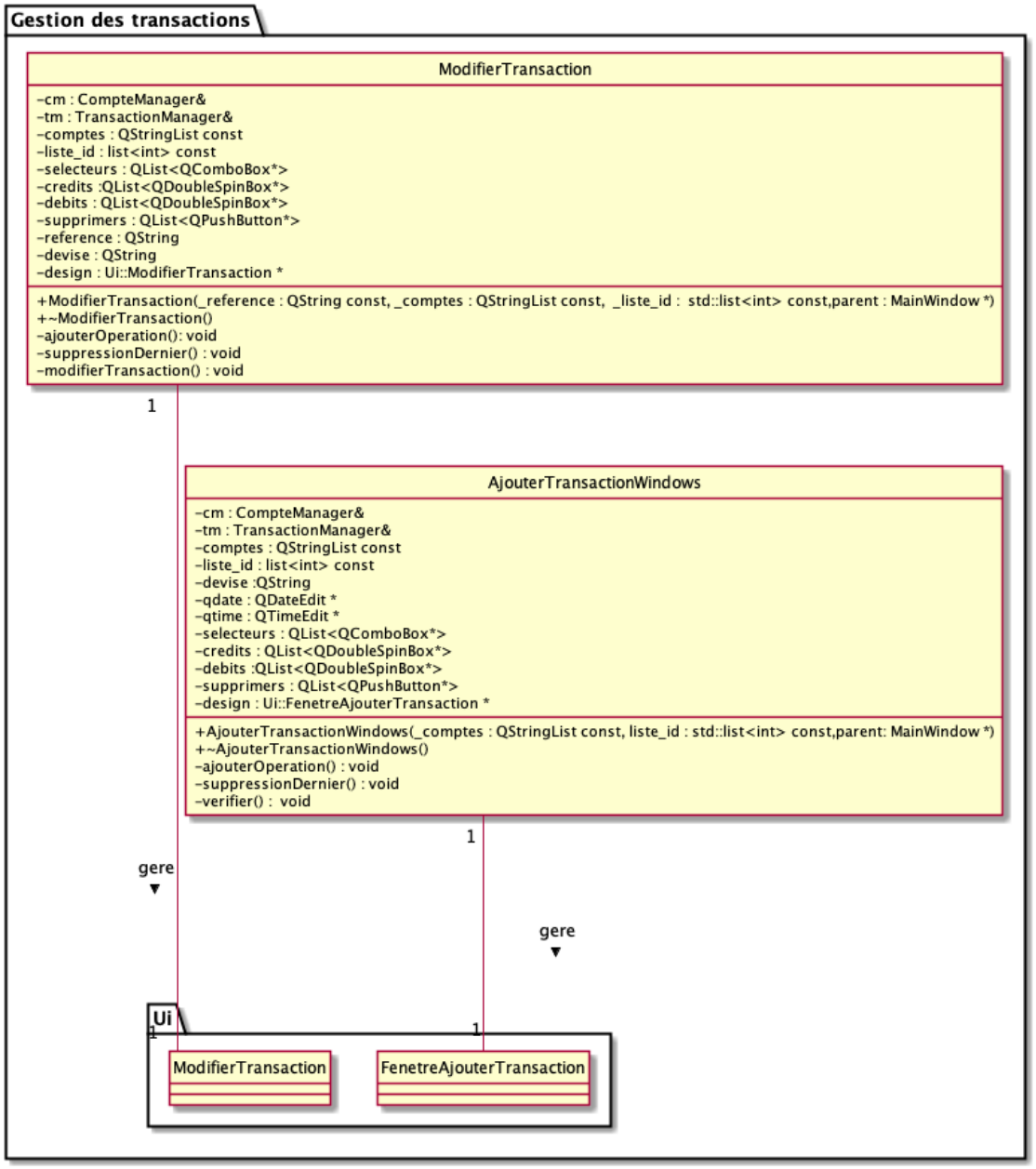

FIGURE 12 – Gestion des transactions

A.2 Présentation des livrables

Ce document est accompagné du code source de l'application, ainsi que d'une vidéo explicative en démontrant le fonctionnement. Une documentation HTML a également été générée avec Doxygen pour plus une explication plus détaillée du rôle de chaque fonction. Le fichier Qt **interface.pro** servant à construire le projet se trouve dans le dossier Interface.