GAYE Amsatou
MINH-GHIA Duong
VIJAYAKUMAR Sahanaa
XU Jiali
ZHANG Chenchao

15/06/2019

Trading Simulator

Le simulateur de Trading

Projet dans le cadre de LO21 P2019
Chargé de TD : Antoine Jouglet

TABLE DES MATIÈRES

INTRODUCTION

« Trading Simulator » est un projet scolaire dans le cadre de l'UV LO21, programmation orientée objet. Le projet consiste à concevoir une application de simulation de trading sur des devises. Elle permet de simuler des transactions sur un marché similaire à Forex[1] : l'application possède trois modes (manuel, pas à pas et automatique) pour simuler des transactions sur une paire de devises par rapport à ses cours sur une période.

L'objectif principal de ce projet est l'apprentissage et l'application des concepts de la programmation objets grâce au langage C++ et l'application Qt. Le projet possède d'autres objectives assez importants tels que la prise en charge d'un projet en équipe, l'apprentissage de Qt…

Remarques :

L'exécution de l'application fonctionne correctement sur une version de Qt supérieur à 4.0 qui possède le module "Chart".

Le code couleur suivant est utilisée le long du rapport pour faciliter la lecture :

- Une **classe** avec ses attributs et ses *méthodes* (constructeurs, destructeurs, slots et signaux)
- Une ***fonctionnalité principale*** avec les *fonctionnalités dérivées* et les boutons mentionnés

[1] Forex : marché d'échange de devises convertibles (https://fr.wikipedia.org/wiki/Forex)

I. DESCRIPTION DES FONCTIONNALITÉS

L'application dispose de nombreuses fonctionnalités qui peuvent être manipulées via les widgets de l'interface. Ces fonctionnalités seront ici décrites dans l'ordre d'utilisation de l'application.

Lors du lancement de la simulation, la fenêtre principale s'ouvre sur une liste de simulation, composée de simulations sauvegardées, avec trois options : ***ouvrir une nouvelle simulation***, ***créer une nouvelle simulation*** ou ***charger une simulation sauvegardée***.

I. A. LANCER LA SIMULATION

Le bouton continuer ouvre une première interface demandant de sélectionner une simulation pour la charger ou de commencer une nouvelle simulation.

I. A. 1. OUVRIR UNE NOUVELLE SIMULATION

Le bouton + en haut de la fenêtre permet d'ouvrir une autre page pour faire une autre simulation. Le fonctionnement est similaire à des onglets sur une fenêtre de navigation : il est ainsi possible de ***mener plusieurs simulations en même temps***. Néanmoins, il faut noter que mener de nombreuses simulations en même temps peut faire lagger le programme ou le faire beuguer.

I. A. 2. CHARGER UNE SIMULATION POUR LA POURSUIVRE

Ce premier état de l'interface permet de charger une simulation préalablement sauvegardée : elle permet de charger la série de cours sur laquelle s'est déroulée la simulation, les paramètres associés, les notes prises et les transactions faites. Il faut sélectionner la simulation et appuyer sur le bouton charger la simulation.

I. A. 2. DEMANDER UNE NOUVELLE SIMULATION

La troisième option proposée est la création d'une nouvelle simulation.

Premièrement, il faut *choisir la paire de devises* en sélectionnant la devise de contrepartie parmi une liste de devises dans une liste déroulante et la devise de base de la même façon. Si la devise n'est pas présente dans la liste de devises, il est possible de la ***créer une nouvelle devise*** en appuyant sur le bouton ajouter une devise. Ce bouton ouvre une fenêtre secondaire : pour ajouter une nouvelle devise, il suffit de fournir dans les champs de texte : un code, un nom pour la monnaie et une zone géographique (non obligatoire), et valider. La devise créée sera immédiatement ajoutée dans les listes déroulantes de devises.

Ensuite, il faut *charger une série de cours OHLCV* en sélectionnant un fichier CSV. Attention, l'application ne vérifie pas si le cours sélectionné correspond à la paire de devises en question, la responsabilité revient à l'utilisateur. Le fichier de cours peut être placé dans n'importe quel répertoire accessible.

Puis, le nom de la simulation soit être saisie dans le champs de texte à disposition. Enfin, il faut appuyer sur le bouton correspondant au mode de simulation souhaité (mode manuel, automatique ou pas à pas).

Deuxièmement, il faut *remplir les paramètres de la simulation* : c'est-à-dire fixer le montant de base (0 par défaut), le montant de contrepartie (1000000 par défaut), le broker (0.001 par défaut), la date de début (par défaut, il correspond à la date de la première bougie du fichier de cours chargé) et selon le mode : il faut *sélectionner la stratégie* parmi une liste déroulante de stratégies implémentées (pour le mode automatique), et il est possible de fixer des paramètres concernant les indicateurs RSI et EMA.

Enfin, la simulation est créée en appuyant sur le bouton create.

I. B. SIMULER

Il existe trois modes pour la simulation : le ***mode automatique*** où les transactions sont faites automatiquement par l'application, le ***mode manuel*** et le ***mode pas à pas*** où l'utilisateur fait les transactions.

I. B. 1. FONCTIONNALITÉS GÉNÉRALES

Quelque soit le mode de simulation choisi, les fonctionnalités suivantes seront toujours disponibles sur la fenêtre principale.

Il est possible de ***sauvegarder la simulation*** en cliquant sur le bouton sauvegarder : les différents données de la simulation (paramètres, transactions, chemin au fichier de cours, indicateurs…) sera sauvegardé sous forme de fichiers dans les répertoires du systèmes.

Au centre de la fenêtre est ***affiché le graphique en chandelier de la série de cours*** sur le premier onglet. Cliquer sur une bougie permet *d'afficher les données liées à la bougie* : le prix d'ouverture, le prix de fermeture, le prix le plus haut atteint, le prix le plus bas atteint et le type de bougie. Sur le second onglet ***s'affiche le graphique en barre pour le volume et le graphique pour l'indicateur RSI***.

Il est possible d'***afficher les indicateurs MACD et EMA*** sur le graphique en chandelier en cochant sur les boutons correspondants en haut de la fenêtre à côté du bouton sauvegarder. Décocher les boutons permet de *masquer les indicateurs du graphique*. On peut *modifier les paramètres liés aux indicateurs* grâce aux fenêtres secondaires qui s'ouvrent lorsqu'on coche les boutons pour le EMA et le MACD ; pour le RSI, un bouton d'accès aux paramètres est placé en haut.

Les graphiques sont mis à jour à chaque avancée dans le temps.

L'historique des transactions effectuées est affiché sous forme de tableau en bas à gauche avec les informations suivantes : la décision, les montants totaux disposés par l'utilisateur au moment de la transaction et le ROI. Il est mis à jour automatiquement à chaque action au niveau des transactions.

À droite se trouve la partie pour le contrôle (panneau de contrôle) des transactions et/ou du temps. Cette partie propose des fonctionnalités différentes selon le mode de simulation dans laquelle, on se trouve. Néanmoins, quelque que soit la simulation, il y est ***affiché les informations*** concernant les montants, la devise, le broker, la date et le prix d'ouverture du jour. Il faut noter que par défaut dans le mode manuel et pas à pas, la date et le prix d'ouverture affichés correspondent à la dernière bougie sélectionnée.

Il est possible de ***prendre les notes*** grâce à un éditeur en bas de la fenêtre. Il faut d'abord soit sélectionner la note à modifier dans la liste de notes ou soit en rajouter une nouvelle via le widget de gestion de notes à droite. Ensuite, il est possible d'éditer la note via l'éditeur. La sauvegarde se fait en même temps que la sauvegarde de la simulation.

I. B. 2. LE MODE AUTOMATIQUE

Dans le mode automatique, la stratégie choisie est aussi affichée sur le panneau de contrôle à droite. De plus, un timer permet de ***choisir la vitesse de la simulation***, vitesse à laquelle on souhaite avancer dans le temps pour observer les transactions automatiques faites. Il est aussi possible de *stopper la simulation* en appuyant sur le bouton pause et d'*avancer séquentiellement* en appuyant sur le bouton défiler.

I. B. 3. Le mode Manuel

Le mode manuel permet de ***choisir la date pour laquelle on souhaite faire la transaction*** en cliquant sur la bougie correspondante. Néanmoins, il faut noter que faire une transaction pour un jour passer n'annule pas les transactions faites ce jour et/ou les transactions suivantes. Il n'est possible que d'***annuler la dernière transaction faite***, ainsi pour annuler, par exemple la k-ième transaction faite, il faut annuler les transactions les plus récentes jusqu'à la k-ième (comprise).

Pour ***faire une transaction***, il faut choisir le jour, écrire le montant dans le champs mis à disposition et cliquer soit sur le bouton vente (*pour vendre*) ou soit sur le bouton achat (*pour acheter*).

I. B. 4. Le mode Pas à Pas

Le mode pas à pas fonctionne comme un mix des deux modes précédents. Un timer permet de *contrôler l'avancer dans le temps*. La transaction doit être faite dans le temps alloué, ainsi, s'il y a besoin de "*faire une pause*" pour réfléchir sur la transaction, il faut placer le timer sur un grand intervalle de temps (plus de 1h). Il n'est pas possible d'avancer dans le temps en sélectionnant une date.

Pour ***faire une transaction***, il faut éditer le montant et choisir entre l'*achat* ou la *vente*. Il est possible d'***annuler une transaction*** de la même manière que dans le mode manuel ou en sélectionnant une date et en appuyant sur le bouton go back. Néanmoins, annuler une transaction passée annule toutes les transactions postérieures.

II. DESCRIPTION DE L'ARCHITECTURE

II. A. LE NOYAU

Le noyau[2] peut se séparer en six grandes catégories : les devises, les cours, les indicateurs, les stratégies, les transaction et la simulation.

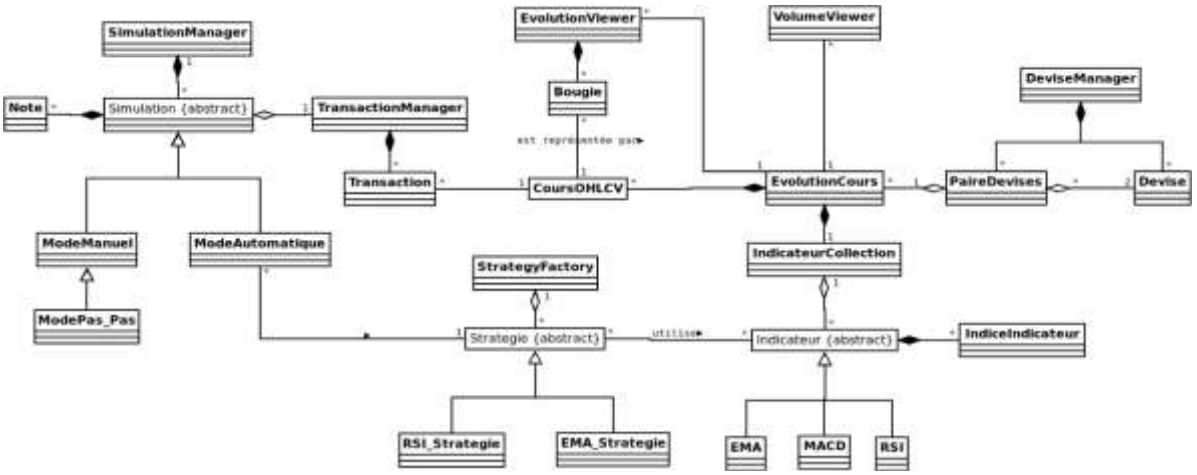

Figure 1:Diagramme de classe du noyau

II. A. 1. LES DEVISES

La classe **Devise** représente une monnaie dans le monde du trading. Elle dispose des accesseurs publiques en lecture pour accéder aux valeurs de attributs, les objets de la classe ne pourront plus être modifiés après leur création. Les constructeurs et les destructeurs sont privés : la création et la destruction des objets **Devise** sera gérée par la classe **DeviseManager**. Cette dernière a accès à l'ensemble de la classe (déclaration d'amitié).

La classe **PaireDevises** représente un couple de monnaies pour lequel la valeur de l'une des monnaie fluctue par rapport à l'autre. Elle possède aussi des accesseurs par lecture mais ne possède qu'un accesseur par écriture : seul le surnom est modifiable. De plus, elle a une méthode publique nommée *toString* pour obtenir une chaine du type "base.contrepartie/contrepartie.base(surnom)".

Les constructeurs et les destructeurs sont privés : la création et la destruction des objets sera prise en charge par la classe **DeviseManager** qui a accès à l'ensemble de la classe (déclaration d'amitié). Pour créer une paire, il faut que les devises, qui la composent, existent, sinon, il faut les créer.

La classe **DeviseManager** est la classe qui gère les classes **Devise** et **PaireDevises** grâce à deux tableaux dynamiques et leurs compteurs associés. Outre les accesseurs en lecture qui permettent d'accéder aux devises et paires où les codes sont passés en paramètres, la classe possède les méthodes suivantes : *getDeviseCodes* pour récupérer la liste de codes des devises existantes et *creationDevise* pour créer une devise.

Le manager devant être unique, le design pattern singleton a été implémenté en incluant la structure *Handler* dans la classe et l'attribut statique handler (Handler). Ainsi, La duplication par affectation ou par recopie sont supprimée et le constructeur et le destructeur sont privés. Ce sont les méthodes statiques *getManager* et *libérerManager* qui permettent d'accéder (si existante) ou de créer (sinon) l'instance unique **DeviseManager** et de le libérer.

[2] Les diagrammes de classes du noyau et de ses parties se trouvent aux annexes 1 à 7.

II. A. 2. LES COURS

La classe **CoursOHLCV** est la classe qui représente les prix d'une paire et le volume échangé au cours d'une journée. En plus des accesseurs en lecture et écriture, la classe est dotée de nombreuses méthodes renvoyant un booléen pour identifier la couleur de la bougie et son "type" (toupie, marteau, pendu…).

La classe **EvolutionCours** rassemble une série successive de cours OHLCV dans un tableau. Elle représente l'évolution des cours de la paire sur une période. La classe contient notamment la collection d'indicateurs associés. Outre les accesseurs en lecture, la classe peut ajouter des cours à la série via la méthode *addCours*. La duplication par recopie et la duplication par affectation sont autorisées.

Le design pattern Iterator est implémenté afin de parcourir les éléments du tableau de cours sans révéler la structure (ici, ce sont des pointeurs sur **CoursOHLCV**). Bien que les cours puissent être gérés indépendamment de cette classe, **EvolutionCours** aura le rôle de manageur car un cours a plus de sens parmi d'autres pour faire du trading.

II. A. 3. LES INDICATEURS

La classe **IndiceIndicateur** est un format de représentation de la valeur d'un indicateur pour un certain jour. Elle possède des accesseurs en lecture et écriture, et une méthode *toString* pour avoir une chaine sous la forme de "date.'Indicateur :'.donnee".

La classe **Indicateur** est une classe abstraite permettant de donner forme aux indicateurs qui vont être ajoutés. Elle est composée d'un tableau de **IndiceIndicateur** afin de répertorier toutes les valeurs d'un indicateur à chaque cours d'une série. Le design pattern Iterator est implémenté afin de parcourir les éléments du tableau. Dans notre application, l'**EMA**, le **RSI**, et le **MACD** sont implémentés comme classe fille de la classe **Indicateur**.

La classe **IndicateurCollection** a accès à la classe **Indicateur** pour pouvoir constituer la liste des indicateurs créés et les stockés dans l'attribut IndicateurDictionnary (QHash<QString, indicateur*>). Elle possède un constructeur privé, un destructeur privé, et un accesseur en lecture. Initialement, les tables de **IndiceIndicateur** pour chaque indicateur est vide : ils sont alloués lors du premier appel par l'utilisateur grâce aux méthodes *generateIndice* ou *setParameters*.

Chaque objet **EvolutionCours** a un objet **IndicateurColleection** associé puisque les indicateurs n'ont de sens que sur une série de cours OHLCV.

II. A. 4. LES STRATÉGIES

La classe **Stratégie** comme la classe **Indicateur** est une classe mère abstraite sur laquelle sur baseront les classes de stratégies de trading. Chaque stratégie à un nom et porte sur un objet **EvolutionCours**. La méthode (virtuelle) *operator()* est celle qui devra renvoyer la décision à prendre et le montant en question : un nombre négatif pour une vente, positif pour un achat et nul en cas d'inaction. Le constructeur et destructeur sont privés car c'est la classe **StratégieFactory** qui gère **Stratégie**.

Pour le moment, elle possède deux classes filles : **EMA_Stratégie**, **RSI_Stratégie**.

La classe **StrategieFactory** a accès à la classe **Strategie** pour pouvoir constituer la liste des stratégies créées et les stockées dans l'attribut strategieDictionnary (QHash<QString, strategie*>). Elle possède un constructeur et un destructeur privés, et un accesseur en lecture. Cette classe est gérée par la classe Simulation puisque les stratégies n'ont de sens que dans une simulation. Les design patterns Singleton et Factory Method ont été appliquée afin que la classe accomplisse son rôle correctement.

StrategieFactory stocke que les prototypes de **Strategie** sous type d'un QHash<QString, Strategie*>. Les stratégies ont besoin d'un objet **EvolutionCours** avec les paramètres des indicateurs sur laquelle elles se basent, pour être utilisées (grâce à la méthode *getStrategie*).

II. A. 5. LES TRANSACTIONS

La classe **Transaction** représente une transaction de montant faite un jour selon le cours OHLCV d'une paire de devises. Elle est conçue de telle sorte que les transactions forment une liste chainée avec les transactions qui s'empilent les unes par-dessus les autres. Outre les accesseurs en lecture, elle possède des méthodes pour parcourir la liste chainée. Le constructeur et le destructeur sont privés car c'est la classe **TransactionManager** qui s'occupera de la classe **Transaction**.

TransactionManager est la classe gérant les transactions lors d'une simulation. Les transactions sont notées par un pointeur sur la tête de la liste en plus des paramètres initiales de la simulation. Le constructeur et le destructeur sont privés car c'est la classe **Simulation** qui s'en charge. Le manager n'a pas été implémenté comme un singleton car il est nécessaire d'avoir accès à plusieurs managers pour faire des simulations en parallèle.

II. A. 6. LA SIMULATION

La classe **SimulationManager** est une classe à instance unique se chargeant des objets de la classe Simulation (des classes filles de Simulation). Elle permet de lister les simulations, d'ajouter des simulations, ou de montrer les simulations sauvegardées ou existantes.

La classe **Simulation** est une classe abstraite sur laquelle se baseront les classes de simulations. Chaque simulation est identifiée par un type (manuel, pas à pas ou automatique) et par un nom. Elle s'applique sur une série de cours OHLCV pour laquelle des transactions sont faites et des notes sont prises. Elle possède des diverses méthodes publiques pour gérer les transactions, sauvegarder les transactions, la simulation ou les cours, ajouter les notes, et charger des transactions ou les notes.

La classe **ModeAutomatique** est un **QObject** héritant de la classe **Simulation** représentant le contexte pour une simulation en mode automatique : c'est-à-dire qu'elle possède un timer de type QTimer* pour gérer le parcours dans le temps et les slots appropriés pour faire passer le message des choix sur le timer au niveau applicatif et au niveau du noyau.

La classe **ModeManuel** hérite de la classe **Simulation** et représente le contexte pour une simulation en mode manuel. Les slots et signaux associés pour faire passer les messages des choix de transactions au niveau applicatif et au niveau du noyau.

La classe **ModePas_pas** hérite de la classe **ModeManuel** et représentant le contexte pour une simulation en mode pas à pas : elle se comporte presque comme la classe Manuel mais avec un timer. Elle contient en plus les slots et signaux appropriés pour traiter les choix sur le timer au niveau applicatif et au niveau du noyau.

La classe **Note** contient les notes prises à une certaines dates pouvant être modifiées dans le temps. Une note n'existe pas en dehors d'une simulation, ainsi les notes sont gérées dans la classe **Simulation** sous forme de QList.

II. B. L'INTERFACE UI

L'UI[3] est structurée en trois grandes parties : la fenêtre principale sur laquelle se dessine l'interface principal, les classes représentant les données graphiquement, et le panneau de contrôle des simulations composant l'interface principale.

II. B. 1. L'INTERFACE PRINCIPALE ET LA FENÊTRE PRINCIPALE

MainWIndow représente la fenêtre principale. Elle est composée d'objet **SimulationTab** et gère son cycle de vie. La classe **SimulationTab** représente les onglets de simulations sur l'interface : elle permet de faire plusieurs simulation en même temps et prend en charge le cycle de vie de l'interface qui la compose. **MainInterface** est la classe principale représentant l'interface de communication avec l'utilisateur, elle fait partie de la fenêtre principale.

La classe **Configuration** s'occupe de la gestion de la création de la simulation : elle demande les paramètres pour la simulations : devises, fichier de cours, nom de la simulation, mode de simulations, montants disposés initialement, les intérêts du broker, la date de début de la simulation…etc. et lance la simulation.

AddDeviseDialog permet la configuration des paramètres liés à la devise : c'est-à-dire que la classe s'occupe de demander les paramètres pour une nouvelle devise et qui se charge de son ajout. De la même manière, la classe **addIndicateurDialog** se charge de configurer les paramètres d'un l'indicateur et de le mettre à jour.

La class **Info** fait partie de **MainInterface**. Son rôle est d'afficher des informations sur l'application, notamment lorsqu'on clique sur le logo trader.

La classe **NoteItem** représente l'interface avec la classe **Note** dans l'interface principale. Son rôle est la mise à disposition de l'éditeur de note avec la table de gestion des notes.

AddIndicateurDialog est la classe qui gère la configuration des paramètres de l'indicateur sélectionné. Ainsi, la classe a à disposition des méthodes, des slots et des signaux afin de modifier la configuration d'un indicateur, de recalculer l'indicateur et de faire les mises à jour nécessaires sur la fenêtre principale

III. B. 2. LES GRAPHIQUES

La classe **Bougie** représente graphiquement un cours OHLCV sous forme de bougie. Elle est donc associée à un objet **CoursOHLCV** : la taille de la bougie dépend du prix d'ouverture et du prix de fermeture, la taille de ses mèches dépend du prix le plus haut et du prix le plus bas du cours, et sa couleur dépend de la fluctuation de la journée. La classe comprend des signaux et des slots pour afficher les données d'une bougie au clique de l'utilisateur et pour afficher la bougie sur le graphique.

La classe **EvolutionViewer** se charge d'afficher le graphique en chandelier et, les indicateurs EMA et MACD. Puisqu'elle est composée de bougies, elle est responsable du cycle de vie des objets Bougie (analogie à **EvolutionCours** responsable de la classe **CoursOHLCV**). La classe dispose de signaux et de slots pour mettre à jour le graphique lorsqu'on avance ou recule dans le temps ou lorsque l'utilisateur se déplace sur le graphique avec la barre de scroll.

On a choisi de représenter les indicateurs MACD et EMA sous forme de séries sur le même graphique car les axes sont les mêmes.

[3] Les diagrammes de classes pour l'UI et ses parties se trouvent aux annexes 8 à 11.

VolumeViewer est la classe qui se charge de deux graphiques au niveau de l'interface : l'affichage du graphique de volume en barre et le graphique pour l'indicateur RSI. Elle dispose de slots et signaux similaires à la classe **EvolutionViewer** puisque l'affichage des graphiques doit être mise à jour à chaque avancé dans le temps.

II. B. LE PANNEAU DE CONTRÔLE DES TRANSACTIONS

Les classes **modeManuelWidget**, **modeAutowidget**, **ModePasPaswidget** s'occupent de la partie interface du panneau de transaction dans le cas du mode manuel, automatique et pas à pas respectivement. Ainsi, elles sont associées chacune à la classe dont elles représentent l'interface et à un manager pour les transactions. Selon le mode, les classes n'ont pas la même responsabilités :

- La classe **modeAutowidget** ne gère que le timer du côté utilisateur.
- La classe **ModePasPaswidget** doit gérer un timer différent et les transactions du côté utilisateur
- La classe **modeManuelWidget** gère les transactions et le choix du cours du côté utilisateur

On a choisi de ne pas faire une classe abstraite mère ou d'héritages entre ces classes pour faciliter l'implémentation de l'UI.

III. DESCRIPTION DES POSSIBILITÉS D'ÉVOLUTIONS

III. A. UNE ARCHITECTURE SOLIDE ET FLEXIBLE

Notre architecture se base sur les points suivants pour être solide, indépendant, fermé mais ouvert aux évolutions :

- L'indépendance par module : la flexibilité de l'architecture est avantagée par le fait que le noyau fonctionne en parties qui ont une faible dépendance entre elles. Ainsi l'ajout d'un nouveau concept (exemple : une calculatrice qui sauvegarde toutes les opérations) ne brusque pas le noyau.
- Chaque classe du noyau n'a qu'une et unique responsabilité dans le programme.
- Les quatre principes de la programmation objet sont appliqués : l'abstraction, l'encapsulation, la décomposition et la généralisation. Ce qui permet l'adaptation à de nouveaux composants.

III. B. DES EXEMPLES D'EXTENSIONS

III. B. 1. AJOUT D'INDICATEUR

Pour rappel, les indicateurs spécifiques héritent tous de la classe mère abstraite **Indicateur**. Cette dernière définit une structure commune pour les indicateurs où l'implémentation détaillée se fait dans les classes dérivées. Ces classes sont gérées et enregistrées par la classe **IndicateurCollection**.

L'ajout d'un nouvelle indicateur se fait facilement au niveau du noyau : il suffit de créer une classe héritant de la classe **Indicateur** et de surcharger les méthodes déclarées dans la classe mère. Ensuite, il faut faire appel au constructeur de la classe qu'on vient d'implémenter dans la classe **IndicateurCollection** pour qu'elle soit prise en compte dans l'application.

III. B. 2. AJOUT DE STRATÉGIE

La partie sur la stratégie est conçue de la même manière que la partie sur les indicateurs. En effet, les classes spécifiques de stratégies héritent de la classe mère abstraite **Strategie**, qui définit la structure commune. L'implémentation est ensuite détaillée dans les classes filles. La classe **StratégieFactory** a un rôle similaire à **IndicateurCollection** dans le sens où elle gère et enregistre les stratégies implémentées. De plus, **StrategyFactory** étant une classe suivant le design pattern Factory et Singleton, elle stocke les prototype des stratégies : il faut faire appel à la méthode *getStratégie* (en fournissant en paramètres le nom de la stratégie et un objet **EvolutionCours**) pour créer véritablement la stratégie

Ajouter une stratégie demande d'implémenter ladite classe comme classe fille de **Strategie**. Il faut nécessairement surcharger la méthode *operation()* puisque c'est elle qui va donner les décisions à prendre vis-à-vis des transactions. De façon analogue aux indicateurs, il faut ajouter un appel au constructeur de la classe ajoutée dans la classe **StrategieFactory** pour que cette dernière reconnaisse l'existence de la stratégie.

III. B. 3. AUTRES ÉVOLUTIONS

De la même manière, il est possible d'ajouter d'autres "types de simulation" sous forme de classes qui hériteraient de la classe **Simulation**.

De plus, tant que les simulations sont exécutées indépendamment, il peut être possible d'exécuter une simulation sur un thread afin d'optimiser les performances du programme.

IV. DESCRIPTION DE L'ORGANISATION

IV. A. LE PLANNING

Le groupe s'est formé le 26 Avril 2019 mais il n'a été actif qu'à partir du 3 Mai à cause des examens. Une réunion chaque semaine était organisée afin de se coordonner dans l'avancement du projet en plus de l'utilisation de réseaux sociaux.

Voici un tableau de présentation du planning :

Le travail à effectuer	Date prévue	Date finale	Les raisons du délai	Remarques
Lecture du projet Idée de conception	03/05/19	03/05/19		
L'architecture du noyau en UML	09/05/19	10/06/19	Ajout d'éléments pendant l'implémentation	Le diagramme de classe conceptuel a été terminée à la date prévue.
Implémentation du noyau	30/05/19	30/05/19		Il y a eu des petites corrections pour une meilleure exécution.
Implémentation de l'interface	10/06/19	13/06/19	Des oublis de fonctionnalités Des problèmes de compilation	
Doxygen	13/06/19	11/06/19		
Vidéo	13/06/19	15/06/19	Des problèmes de coordinations	
Rapport	13/06/19	15/06/19	Les retards dans l'implémentation Les modifications du code	
Le projet	13/06/19	15/06/19	Les retards	

Tableau 1: Planning su projet

Les premières étapes du projet ont été divisées entre les membres du groupe pour réduire la quantité de travail et pour s'attendre le moins possible les uns et les autres, le travail a été partitionné en parties les plus indépendantes possibles.

IV. B. LA CONTRIBUTION

Le groupe a fonctionné en collaboration avec un dépôt commun et le travail a été partitionné autant que possible en part égale. Néanmoins, il faut souligner que M. Minh-Nghia Duong a pris une position de leader et a porté les responsabilités d'un chef d'équipe. En effet, il a managé le groupe et le développement du projet, et il a su diviser le travail correctement pour l'attribuer aux membres de l'équipe. De plus, M. Minh-Nghia Duong s'est occupé de merger nos parties.

Voici le tableau de représentation de la participation (en vert) des membres de l'équipe dans les différentes parties du projet :

Le travail à effectuer	G. Amsatou	M. Duong	V. Sahanaa	Z. Chenchao	X. Jiali
UML					
Noyau - les devises					
Noyau - les cours					
Noyau - les indicateurs					
Noyau - les stratégies					
Noyau - la transaction					
Noyau- la simulation					
Ui/Noyau - la sauvegarde					
UI/Noyau - Analyse bougie					
UI - l'interface					
UI - les graphiques					
UI - les transactions					
UI - contrôle des transactions					
UI - éditeur de texte					
Doxygen					
La vidéo					
Le rapport					

Tableau 2:Contribution de chaque membre dans les parties du projet

CONCLUSION

Le projet Trading Simulator permet d'acquérir de l'expérience dans le parcours pour devenir un ingénieur informatique.

Dans un premier temps, le sujet nous positionne dans une situation réelle de demande de projet presque professionnelle : le monde du trading est un monde populaire où les simulateurs de trading sont très importants pour gagner de l'expérience, ainsi de nombreuses applications de simulation de trading ont été développées.

Dans un deuxième temps, le développement de l'application demande de l'organisation dans le travail. En effet, le planning et la répartition du travail au sein de l'équipe sont capitales pour que le projet se déroule correctement. Le planning permet de surveiller la progression du projet et de s'organiser sur les semaines, et la répartition, égale et réfléchie du travail, permet d'avancer le projet plus ou moins indépendamment en évitant les impasses. La réunion d'équipe chaque semaine n'est pas à négliger puisqu'elle permet aux membres de l'équipe de prendre des décisions communes sur des zones de confusion. De plus, la réflexion collective réduit fortement les erreurs de conception.

Enfin, le projet permet d'appréhender les concepts de la programmation orientée objet et en particulier ses principes. La conception de l'architecture, le noyau en particulier est l'étape la plus importante puisque c'est la solidité et la flexibilité de sa conception et implémentation qui vont permettre le bon déroulement du projet lors des étapes suivantes.

Ainsi, Trading Simulator est un projet pour apprendre à travailler en équipe sur la conception d'application en programmation orientée objet.

ANNEXE

ANNEXE 1 : DIAGRAMME DE CLASSE DE LA PARTIE DEVISE

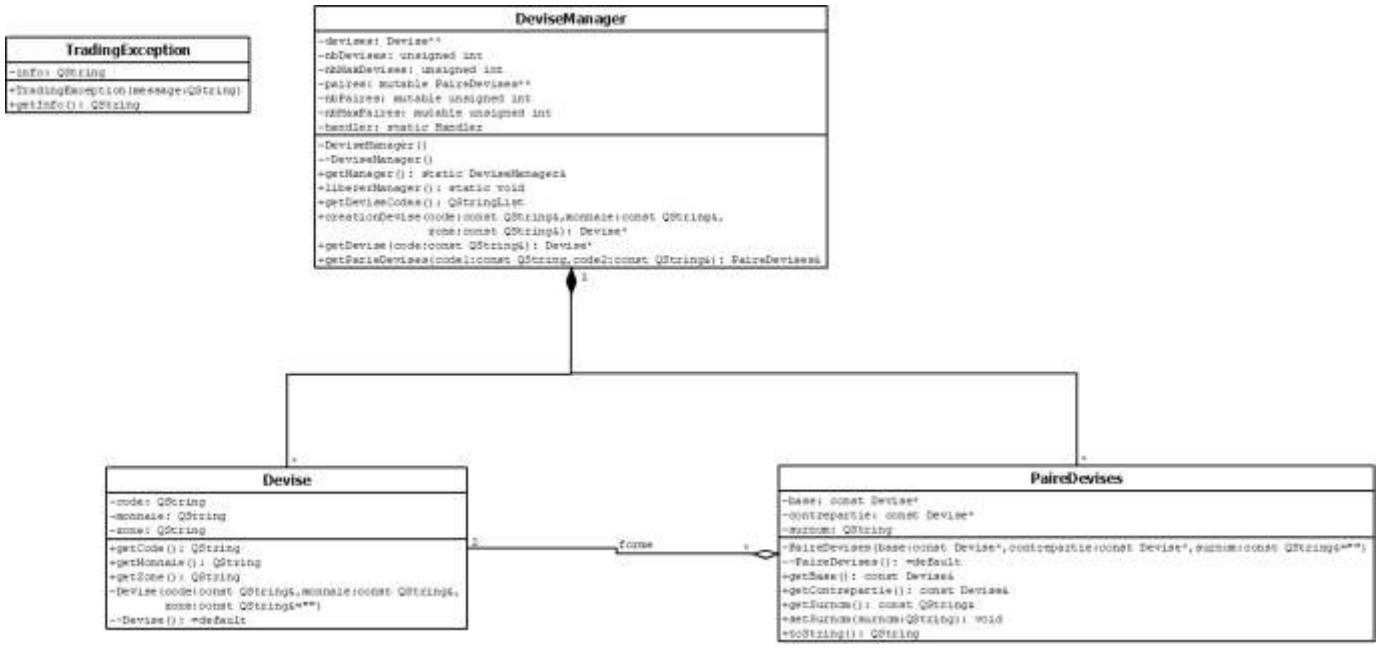

Figure 2: Diagramme de la partie Devise du noyau

ANNEXE 2 : DIAGRAMME DE CLASSE DE LA PARTIE COURS

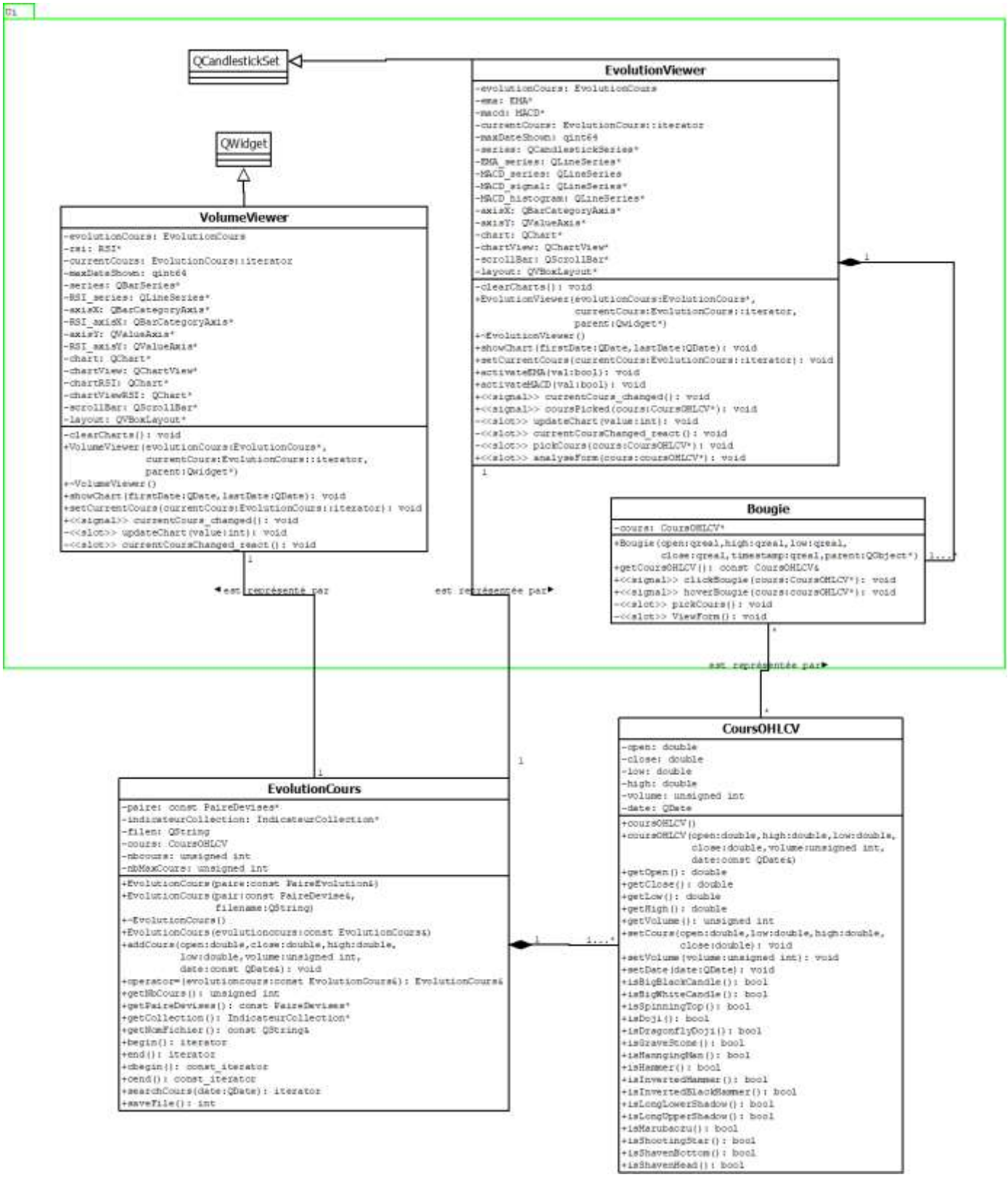

Figure 3: Diagramme de la partie Cours du noyau

ANNEXE 3 : DIAGRAMME DE CLASSE DE LA PARTIE INDICATEUR

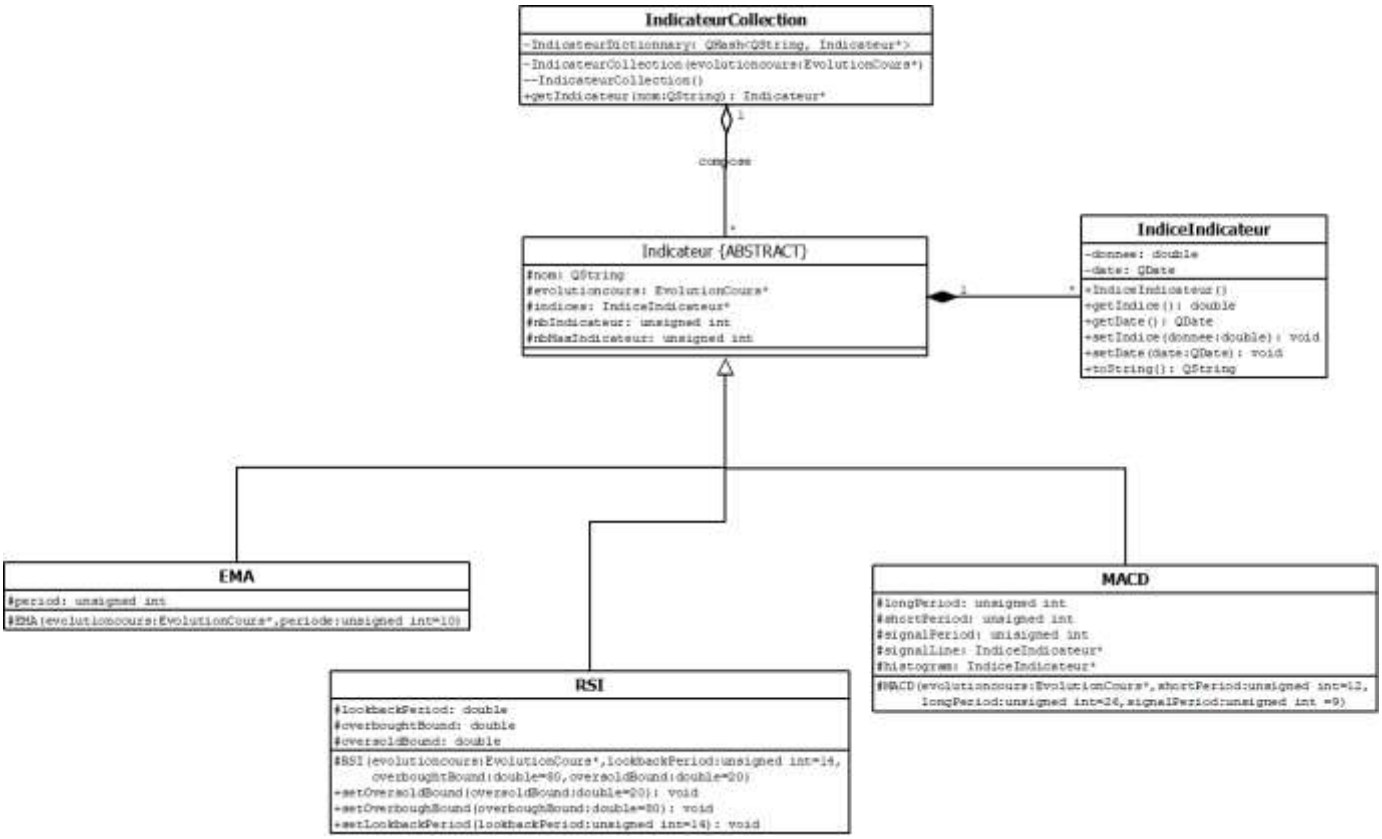

Figure 4: Diagramme de la partie Indicateur du noyau

ANNEXE 4 : DIAGRAMME DE CLASSE DE LA PARTIE SIMULATION

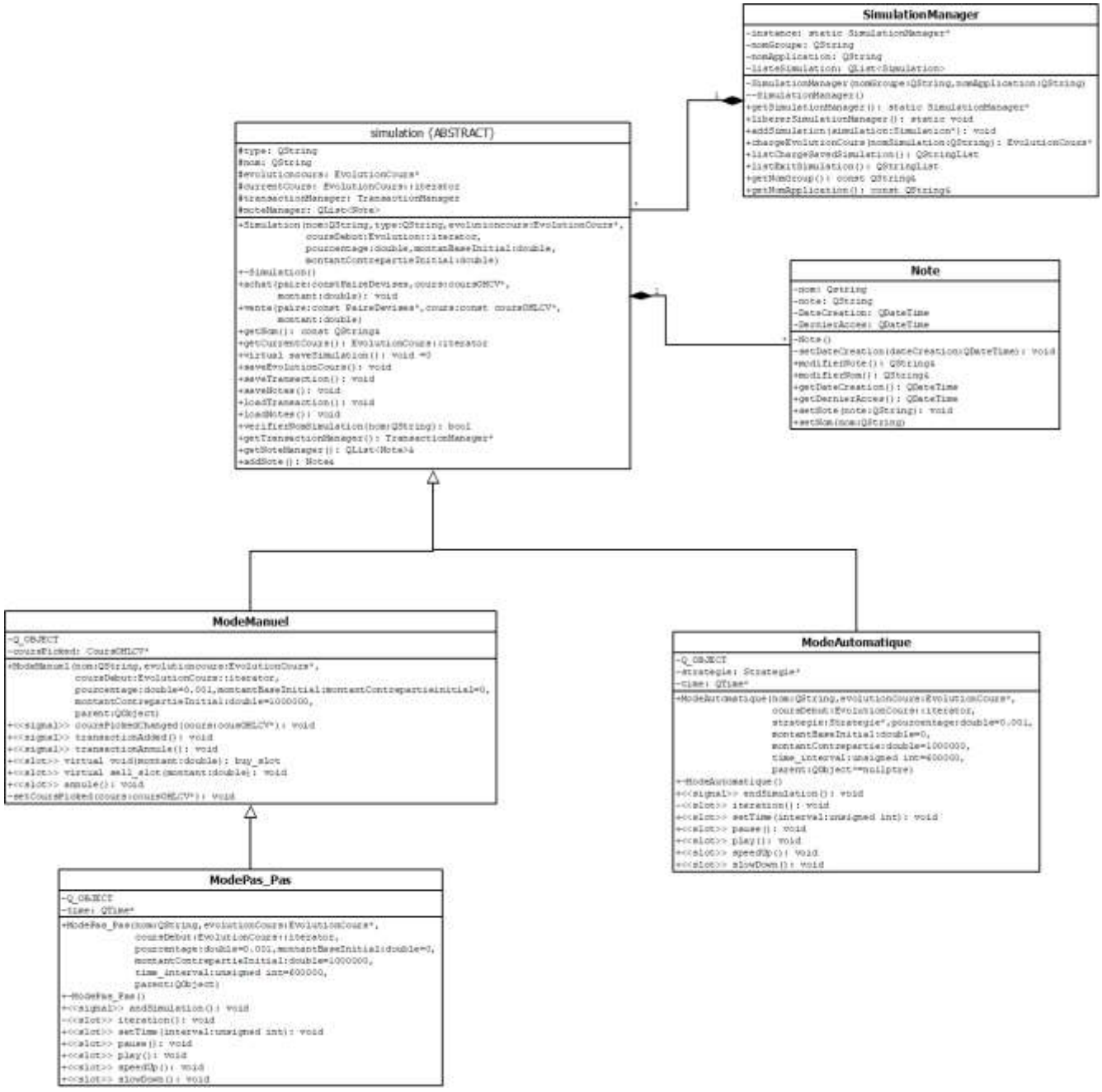

Figure 5: Diagramme de classe de la partie simulation du noyau

ANNEXE 5 : DIAGRAMME DE CLASSE DE LA PARTIE STRATÉGIE

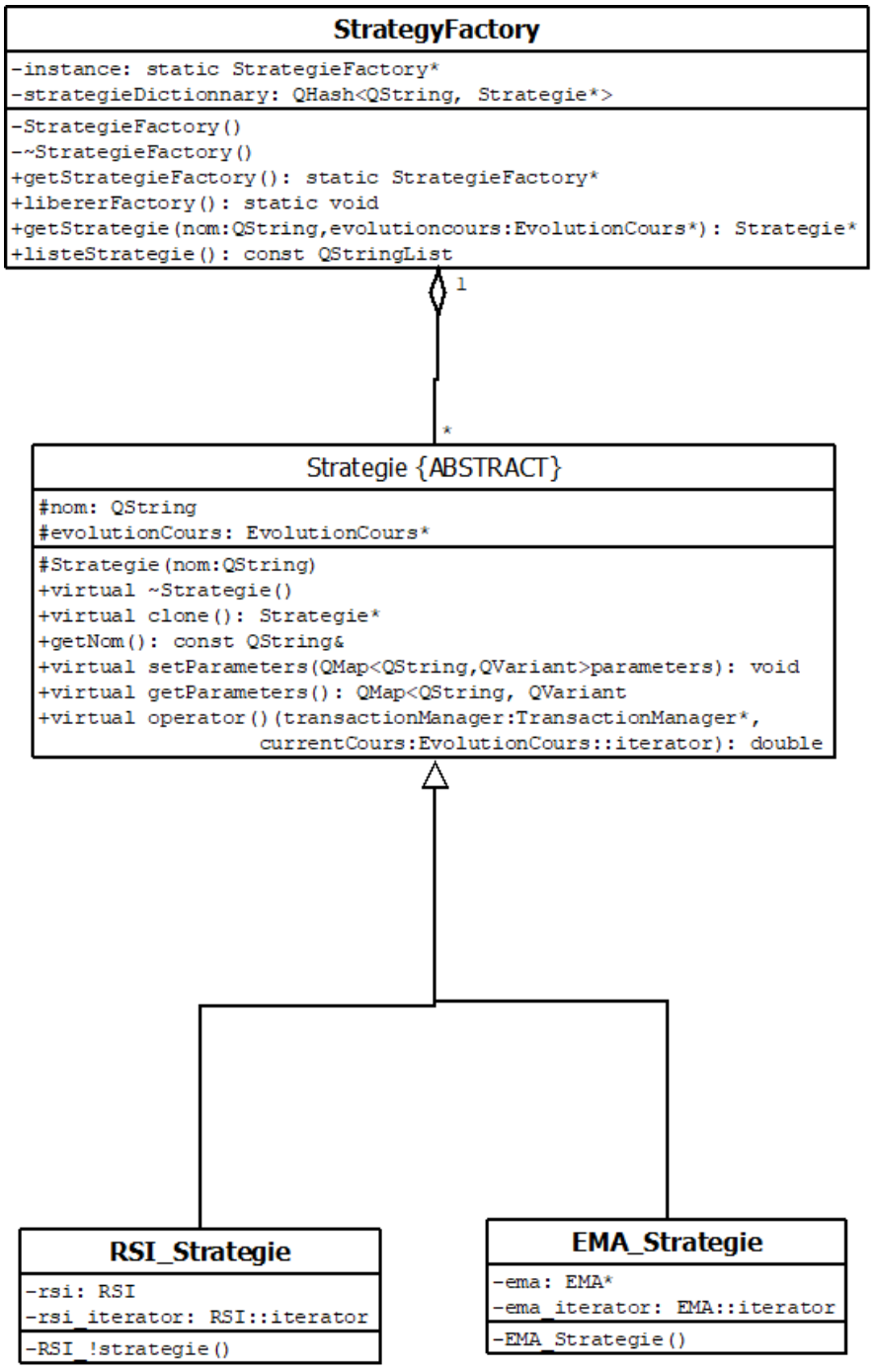

Figure 6: Diagramme de la partie Stratégie du noyau

ANNEXE 6 : DIAGRAMME DE CLASSE DE LA PARTIE TRANSACTION

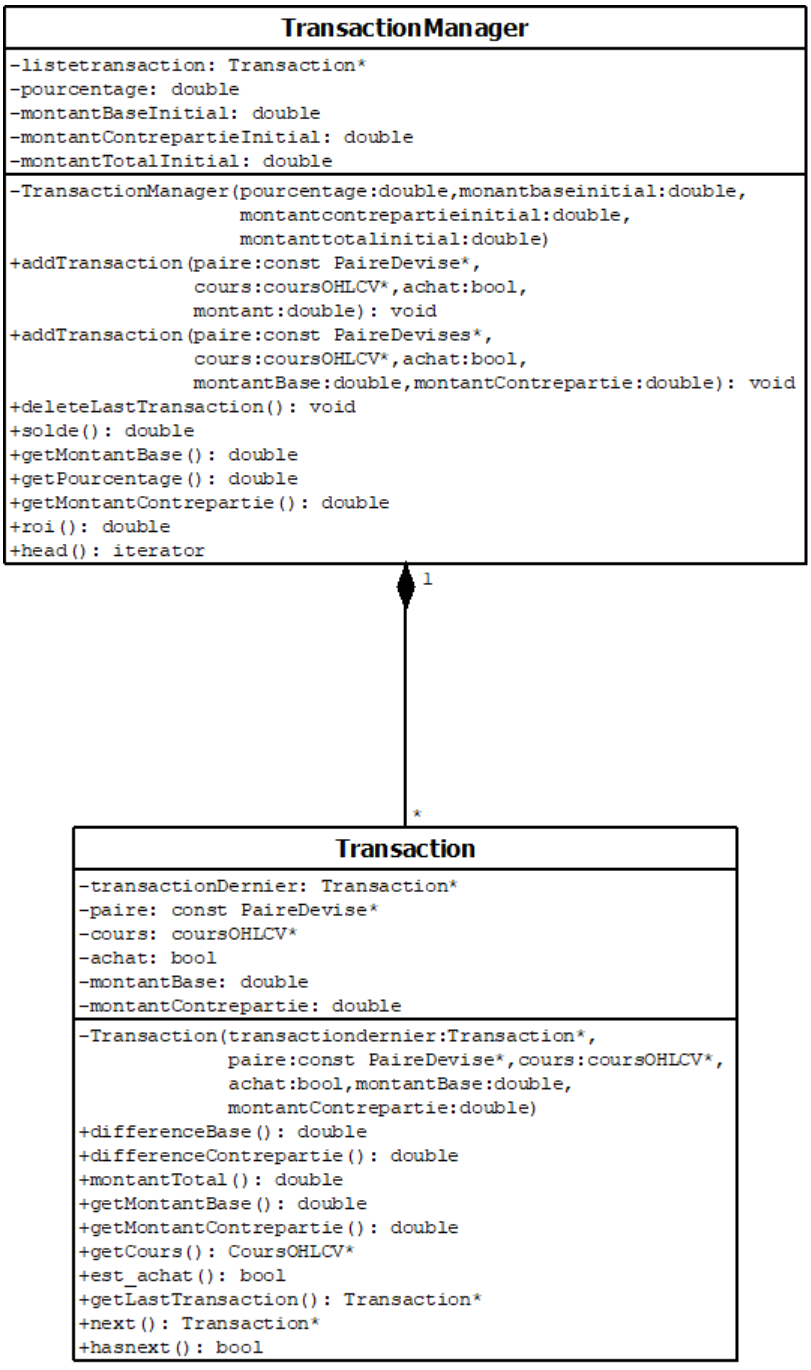

Figure 7: Diagramme de classe de la partie Transaction du noyau

ANNEXE 7 : DIAGRAMME DE CLASSE DE L'UI

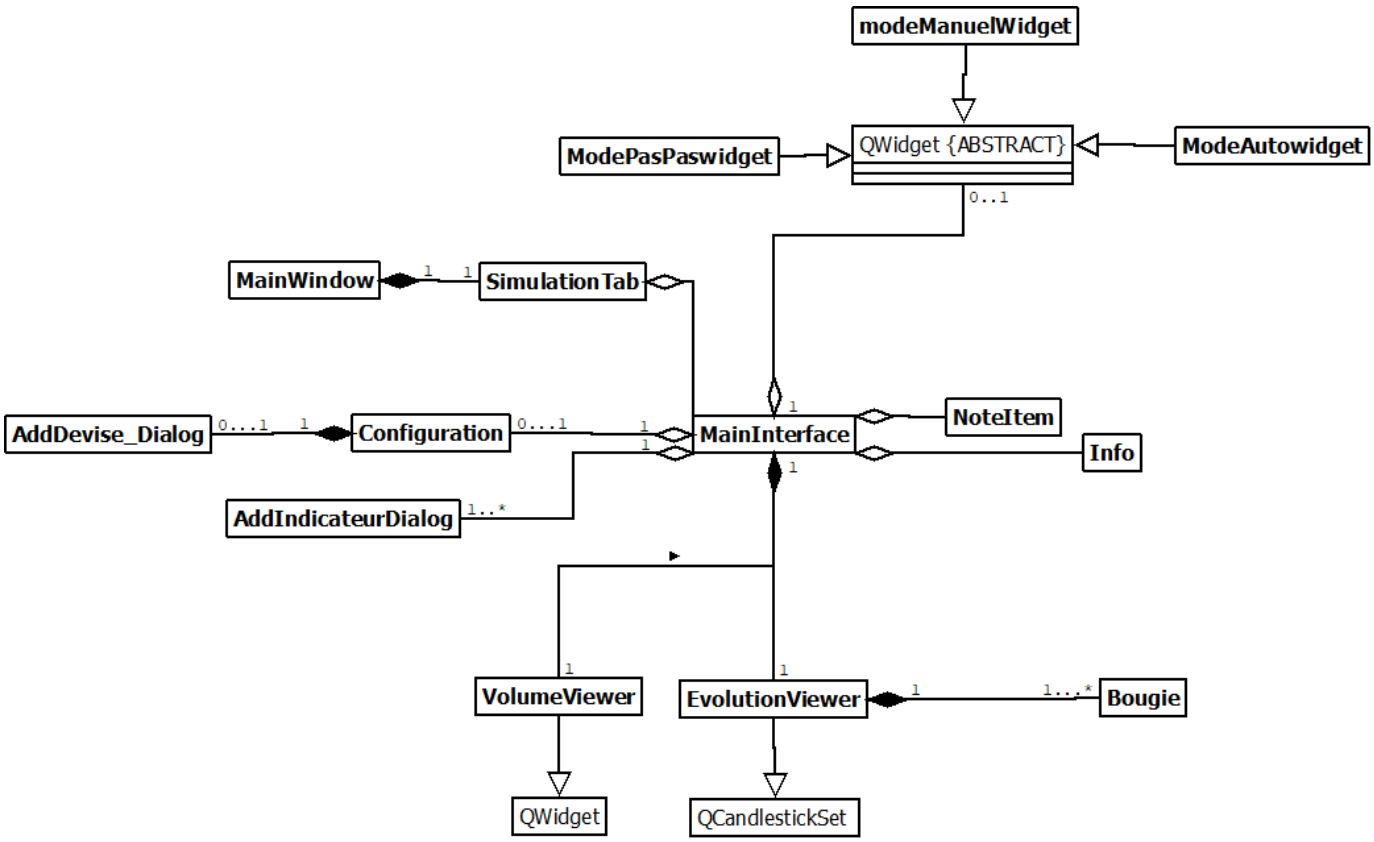

Figure 8:Diagramme de classe de l'UI

ANNEXE 8 : DIAGRAMME DE CLASSE DE LA PARTIE "INTERFACE PRINCIPALE"

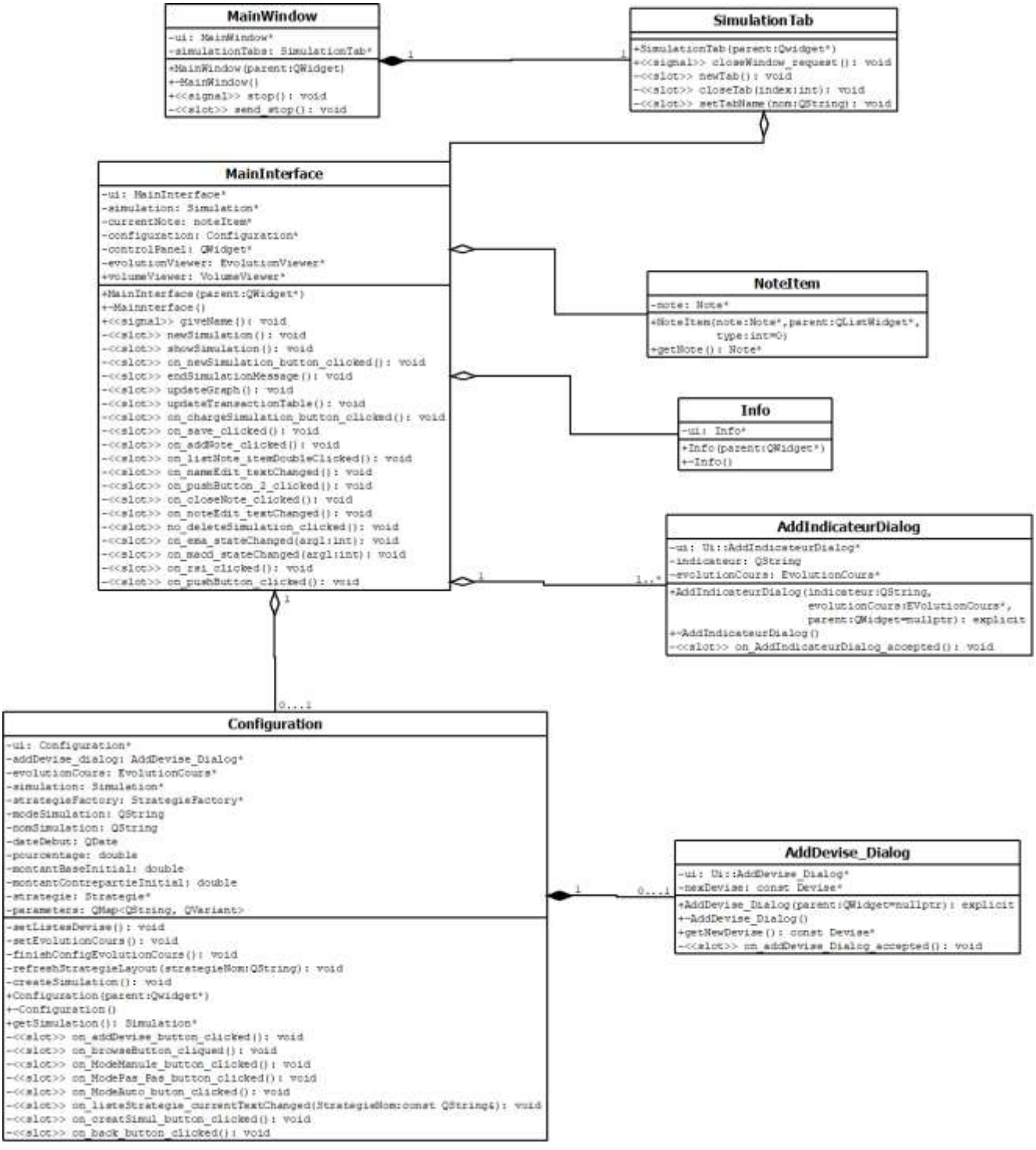

Figure 9: Diagramme de classe de la partie Interface principale de l'UI

ANNEXE 9 : DIAGRAMME DE CLASSE DE LA PARTIE "GRAPHIQUE"

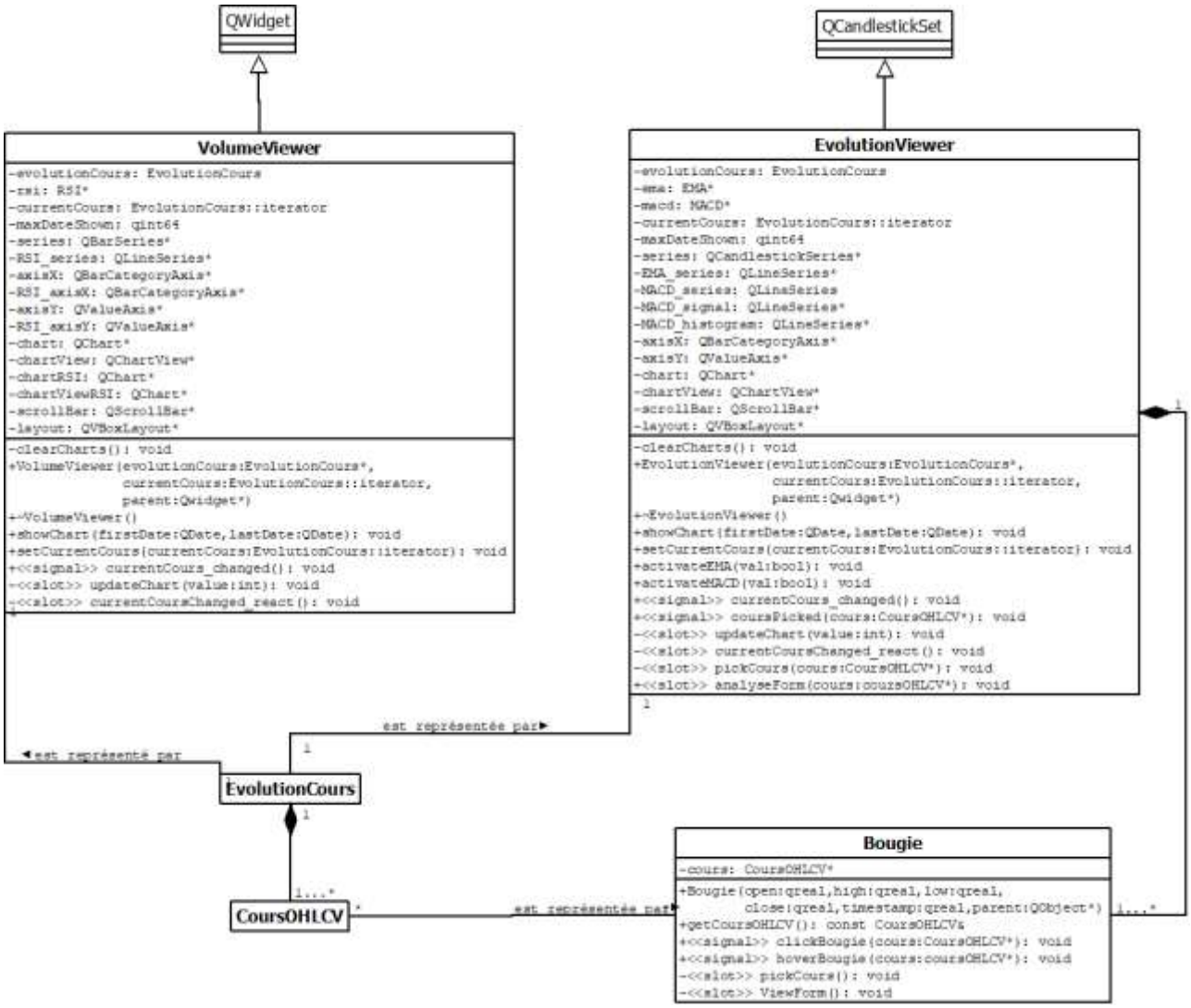

Figure 10:Diagramme de classe de la partie Graphique de l'UI

ANNEXE 10 : DIAGRAMME DE CLASSE DE LA PARTIE "WIDGET POUR LES MODES"

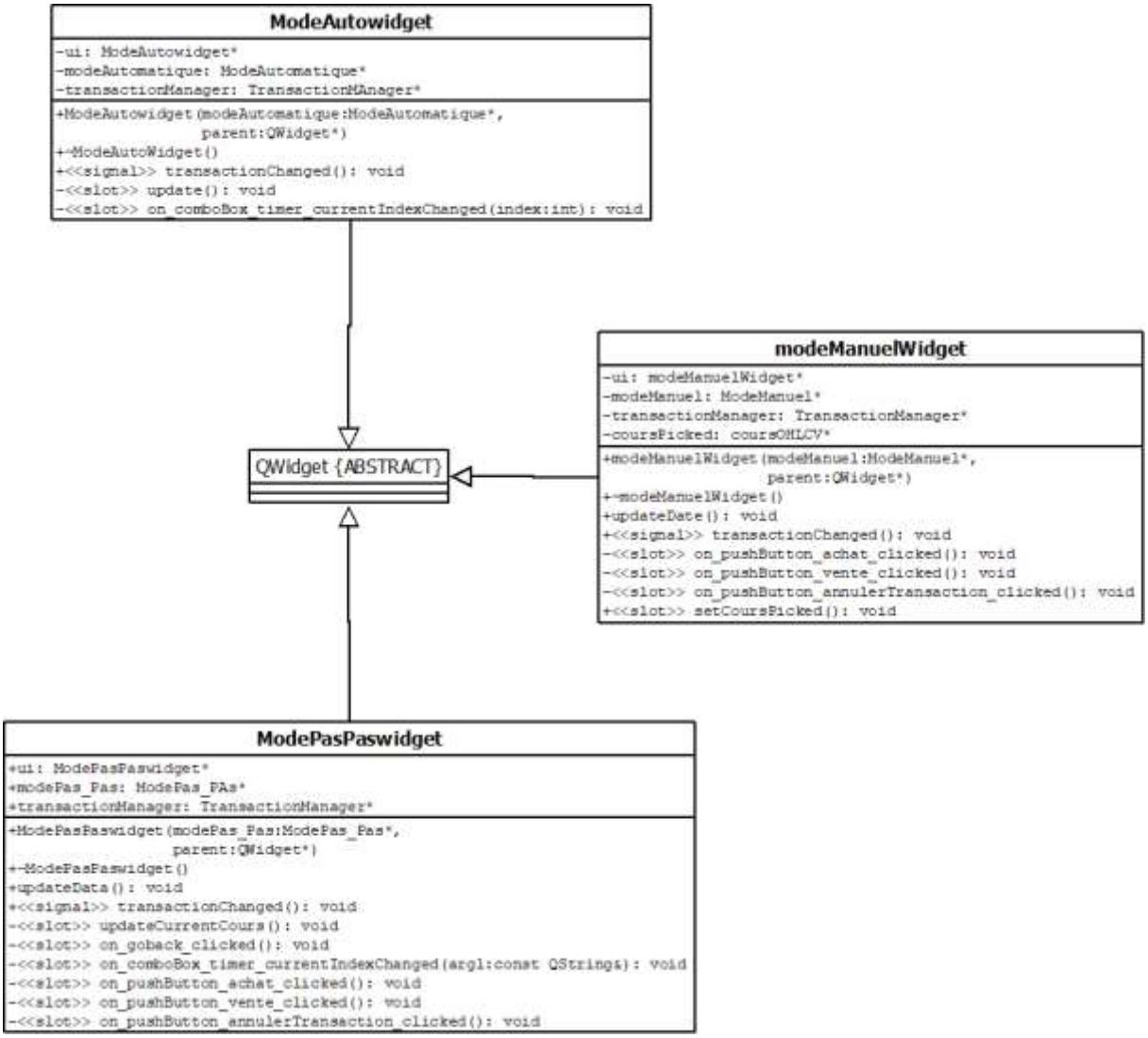

Figure 11: Diagramme de classe de la partie Transaction de l'UI