Rapport LO21 2019
Projet Trading Simulator

ROMON Pierre
LECREUX Tom
JUSZCZAK Eloi
GIRAUD Edmond

Semestre P19

Sommaire

Introduction **2**

Description du contexte **3**

Rappel sur le trading de devises 3

Les données 3

Description de l'architecture **4**

Bloc Devises 5

DeviseManager 5

Description des blocs 6

Architecture pour les transactions 6

Classe TransactionManager 6

Classe Transaction 6

Architecture pour le mode automatique 6

Architecture pour le mode pas à pas 8

Classe TransactionViewer 8

Travail sur la persistance des données : 9

Evolution possible de l'application **9**

Organisation pour l'implémentation de l'application **10**

a) Planning et répartition des tâches 10

b) Organisation du travail et problèmes rencontrés 11

Conclusion : **11**

Introduction

Dans le cadre de l'UV LO21, il nous a été demandé de concevoir et de développer une application de simulation de trading.

Cette application a pour but de permettre à l'utilisateur de s'exercer au trading sur le marché des devises, il aura la possibilité d'effectuer lui-même des transactions fictives d'achat ou de vente.
Elle devra aussi lui permettre de tester différentes stratégies automatiques, pour comparer leur efficacité sur les différentes paires de devises.

Description du contexte :

1) Rappel sur le trading de devises

Notre application s'intéresse au trading sur le marché des devises. Les échanges se feront sur différentes paires de devises en fonction des fluctuations du cours pour ces devises. Une paire de devise dispose d'un cours OHLC qui correspond à son taux de change, ce cours évoluant quotidiennement.

A titre d'exemple la notation EUR/USD 1.2500 signifie qu'il est possible d'échanger 1 euro contre 1.2500 dollars américains.

2) Les données

Pour notre application, l'unité de temps utilisée pour la cotation des paires de devise est la journée.
Sur une journée, la cotation associée à une paire de devises peut connaître des variations. Ainsi, pour appréhender ces variations, il sera possible de consulter 4 données significatives, qui correspondent au cours OHLC :

- Le prix d'ouverture, c'est-à-dire le prix auquel s'est effectué le premier échange de la journée
- Le prix le plus haut, c'est-à-dire le prix le plus haut que la cotation a atteint dans la journée
- Le prix le plus bas, c'est-à-dire le prix le plus bas que la cotation a atteint dans la journée
- Le prix de fermeture, c'est-à-dire le prix auquel s'est effectué le dernier échange de la journée

Ces données peuvent être représentées sous la forme de bougies, dans un graphique appelé chandelier. Le but de cette représentation est de pouvoir représenter les variations d'un cours lors d'une journée afin d'en faire une analyse technique. Un graphe de volume est également disponible pour modéliser la quantité d'échange quotidien effectué pour la paire considérée.

La forme d'une bougie se base sur les données du cours OHLC correspondant:

- La taille corps de la bougie indique si la variation entre le prix d'ouverture et le prix de fermeture. Le corps peut être de 2 couleurs différentes : la couleur rouge si il s'agit d'une baisse ou bien la couleur verte pour une hausse.
- La mèche du haut indique le prix le plus haut de la journée.
- La mèche du bas indique le prix le plus bas de la journée.

Différents indicateurs (RSI, SMA,etc), prenant en compte un ensemble de cours OHLC seront utilisés pour implémenter des stratégies de trading.

3) Fonctionnalités de l'application

Notre application permet à l'utilisateur de pouvoir s'exercer au trading. Elle se divise en 2 modes d'utilisation : le mode manuel et le mode automatique.

Dans le mode manuel, l'utilisateur rentre lui-même ses transactions "jour par jour" pendant une période de temps qu'il aura choisi : il effectuera des achat ou des ventes avec la quantité qu'il souhaite. Il aura la possibilité d'effectuer plusieurs transactions par jour.
Un graphique des bougies des cours OHLC sera affiché à l'utilisateur.
Ce graphique ne lui permettra de voir que les bougie entre la date de début du fichier csv et la date du "jour courant".

En mode manuel, l'utilisateur peut sauvegarder une simulation en cours et la reprendre plus tard à l'endroit où il s'était arrêté s'il le souhaite.
Pendant la simulation, l'utilisateur peut annuler la journée en cours s'il le souhaite, cela aura pour effet d'annuler les transactions effectué pendant le jour courant. Le graphique sera également mis à jour.

En mode automatique, l'utilisateur choisi une stratégie parmi une liste, qu'il appliquera sur une paire de devise pendant une période de temps qu'il aura choisi. L'algorithme de la stratégie effectuera des transactions sans intervention de l'utilisateur selon la spécificité de la stratégie. L'utilisateur n'aura alors pas la possibilité d'effectuer lui même des transactions.

Un bilan des transaction sera affiché graphiquement à l'utilisateur (dans le cas du mode pas à pas, il sera mis à jour à chaque nouvelle transaction ou annulation).

L'utilisateur aura la possibilité de prendre des notes dans un éditeur de texte sur une stratégie, qui sera intégré à la fenêtre. Lors de la sauvegarde d'une simulation, les notes seront sauvegardées aussi, de même lors du chargement.

Description de l’architecture :

Une description détaillée de l’ensemble des fichiers et des classes est disponible en html avec Doxygen.

Cette partie sera consacré à la description des comportements globaux des différents blocs de l’application et on détaillera les choix architecturaux qui y sont associés.

Bloc Devises :

Classe DeviseManager:

Cette classe a été implémenté avec le design pattern singleton ce qui nous permet d'instancier une seule fois cet objet.
L'intérêt de cette classe est de centraliser la construction/destruction des objets Devise et PaireDevise. Ceci permet de n'avoir qu'un seul objet déclenchant toutes les créations et destructions d'objets relatifs aux Devise.

Classe Configuration :

Cette classe correspond à la fenêtre qui apparaît lorsque l'on lance une nouvelle simulation. Cette dernière permet d'afficher les devises disponibles. L'utilisateur pourra donc choisir la devise de base, et la devise de contrepartie qu'il souhaite pour sa simulation, parmi la liste proposée ou encore créer une nouvelle devise, si elle n'est pas disponible dans la liste.

Ensuite, il choisira s'il souhaite lancer sa simulation en mode manuel ou automatique (auquel cas il choisira une stratégie parmi une liste proposée).

Classe Devise & PaireDevise :

Nous avons très simplément implémenté 2 classes : Devise et PaireDevise, pour stocker les informations relatives aux différentes devises / paire de devises.

Classe CoursOHLC :

Chaque cours OHLC pour un jour donné sera un objet CoursOHLC, où seront stocké en attributs toutes les données nécessaires.

Les objets CoursOHLC sont créés lors de l'appel à la méthode de chargement du fichier CSV.

Classe EvolutionCours :

Cet objet nous sert à stocker l'ensemble des cours OHLC. Afin de compléter l'information apportée par les cours OHLC, nous avons implémenté des méthodes correspondantes aux différents indicateurs techniques. L'implémentation sous forme de méthode facilite leur utilisation ainsi que l'ajout de nouveaux indicateurs dans un cas plus complexe, garantissant à notre application une certaine modularité.

Architecture pour les transactions:

Afin de permettre à l'utilisateur en mode manuel, et aux stratégies en mode automatique,de réaliser des opération d'achat ou de vente, nous avons implémenté plusieurs classes permettant la gestion de ces transactions.

Classe TransactionManager :

Comme pour la gestion des devises, nous avons utilisé un design pattern Singleton sur la classe TransactionManager. Cette dernière ne pourra donc jamais être instancié plus d'une fois.
Tout comme pour les devises, cette classe est responsable de la création de transaction, mais elle contient aussi comme attributs la liste de toutes les transactions effectuées lors du mode déclenché.
Cette classe a donc pour but de centraliser les informations et l'utilisation des méthodes relatives aux transactions.

Classe Transaction :

Cette classe correspond à une transaction réalisée à un instant T. Elle possède en attributs toutes les informations nécessaires telles que la date à laquelle elle a été réalisée, la quantité achetée ou vendue, etc.

Architecture pour le mode automatique:

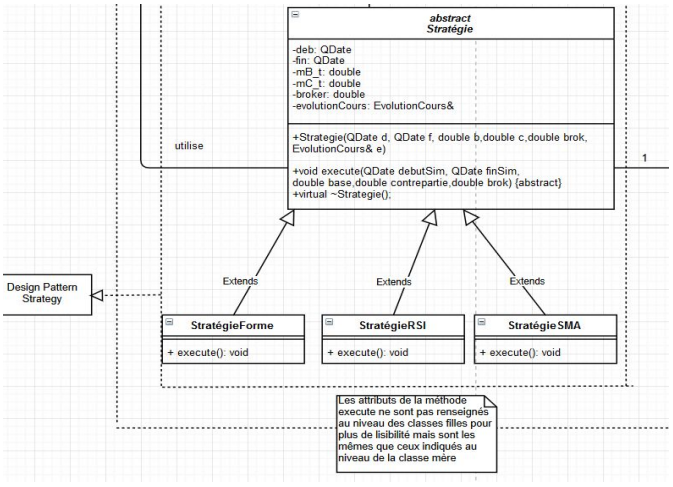
Le mode automatique doit permettre à l'utilisateur d'appliquer, sur le fichier CSV chargé, différentes stratégies de trading, internes au logicielle, afin de les confronter et d'évaluer leur performance.

Nous avons choisi d'implémenter trois stratégies différentes, deux portées sur les indicateurs techniques RSI et SMA et une basée sur l'analyse de la forme des bougies du chandelier.

Deux designs patterns nous ont permis d'implémenter les stratégies.

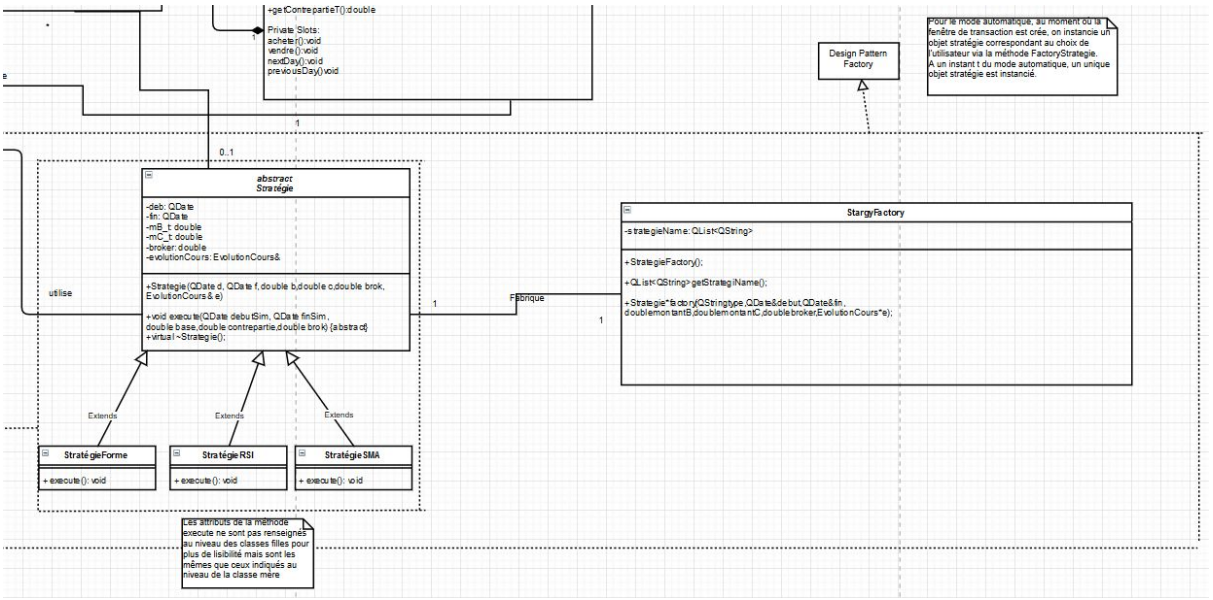
- Strategy
- Factory

De notre point de vue, les stratégies peuvent conserver une structure similaire mais divergent dans leur comportement. Nous avons donc choisi d'implémenter un classe mère abstraite disposant d'une méthode virtuelle pure qui constitue "l'algorithme" a exécuté selon la stratégie choisie. Cette méthode devait donc être défini différemment dans chacune des classes stratégie fille.



Un autre enjeu de cette partie du code réside dans la création des objets stratégies. En effet ces derniers devaient être créés à la volé selon le choix de l'utilisateur. Nous nous somme donc inspiré du design pattern factory en créant un méthode que nous avons appelée *FactoryStratégie* au niveau de notre fenêtre de stratégies. Cette méthode récupère le choix de l'utilisateur quand à la stratégie choisie et instancie l'objet Stratégie correspondant (StrategieForme, StrategieRSI, StrategieSMA).

Nous n'avons implémenté que trois stratégies relativement simples mais dans un cas plus complexe cette architecture simplifie la création de nouvelles stratégie dans le but d'enrichir le modèle de trading.
En effet, afin de créer un nouvelle stratégie, on fait simplement appel au constructeur de la classe mère en définissant la méthode execute selon l'algorithme de notre choix puis il suffit de compléter la condition de vérification de la méthode FactoryStratégie pour instancier la nouvelle stratégie.



Une séquence de trading en mode automatique se déroule donc de la manière suivante: l'utilisateur choisi une stratégie ainsi que les dates de simulation et le montant de devise de base et de contrepartie dont il dispose.
Ces informations sont passée au constructeur de la classe StartegyFactory qui récupère la stratégie sélectionnée par l'utilisateur et instancie l'objet stratégie fille correspondant à la volée puis appelle à la suite la méthode execute de cette stratégie fille.

Des objets transactions sont alors générés via le singleton transactionManager en fonction des instructions de l'algorithme execute puis ces transactions sont ajoutées à la liste des transactions et affichées au niveau de l'objet TransactionViewer (on aura dans ce cas désactivé les boutons d'achat, de vente ainsi que jour suivant et jour précédent de la fenêtre).

Architecture pour le mode pas à pas:

Classe TransactionViewer :

Cette classe correspond à la fenêtre graphique à partir de laquelle l'utilisateur peut décider d'acheter ou de vendre de la devise. Nous y avons donc intégré plusieurs composants graphiques permettant à l'utilisateur de visualiser l'état actuel de sa simulation :

- en haut à gauche, il peut voir quel est le jour courant dans la simulation, la paire de devise sur laquelle il travaille, le montant de la devise base et de contrepartie dont il dispose, le prix d'ouverture de la journée

- en bas à gauche, il peut voir les bougies des cours OHLC allant de la première date du fichier csv jusqu'au "jour actuel" de la simulation. Celui-ci se met à jour lorsque l'utilisateur décide d'annuler la journée en cours ou bien de passer au jour suivant.

- à droite, il a à sa disposition un récapitulatif des transactions qu'il a effectué pendant la simulation.

Dans cette fenêtre nous avons également mis à disposition de l'utilisateur un bouton "jour précédent" qui lui permet de retirer de la liste des transactions les transactions effectuées lors de la journée en cours.

Travail sur la persistance des données :

Nous avons choisi de gérer notre persistance de données en utilisant XML. On sauvegarde les nouvelles devises créées par l'utilisateur. Les simulations en mode pas à pas sont également sauvegardables à tout instant et peuvent être rechargées depuis la fenêtre du mode pas.

Evolutions possible pour l'application

Un des objectifs principaux, au delà de la couverture des différentes fonctionnalités demandées était de pouvoir rendre l'application la plus modulaire possible.
Nous avons tenté au mieux de rendre compte de ce principe de modularité à travers différents choix de conception.

Tout d'abord concernant la possibilité d'ajouter de nouvelles devises/paires de devises: leur création et destruction étant centralisée au niveau de notre singleton DeviseManager, cette fonctionnalité en est alors simplifiée.

Il en va de même avec l'ajout de nouvelles transaction que nous avons également facilité grâce à la création d'une classe singleton TransactionManager.

Concernant les indicateurs OHLC, nous avons choisi de les implémenter comme des méthodes relatives à des cours OHLC et donc centralisées et directement disponibles au niveau de notre objet EvolutionCours. Au delà du simple fait de faciliter leur utilisation, ce choix permet également de générer facilement de nouveaux indicateurs comme méthodes de la classe EvolutionCours.

L'utilisation des designs patterns Strategy et Factory pour les stratégies de trading de l'application permet également de garantir un certaine modularité dans l'ajout de nouvelles stratégies.
Nous avons donc choisi d'implémenter un classe mère abstraite disposant d'une méthode virtuelle pure qui constitue "l'algorithme" a exécuté selon la stratégie choisie. Cette méthode devait donc être défini différemment dans chacune des classes stratégie fille.
En effet, afin de créer un nouvelle stratégie, on fait simplement appel au constructeur de la classe mère en définissant la méthode execute selon l'algorithme de notre choix puis il suffit de compléter la condition de vérification de la méthode FactoryStratégie pour instancier la nouvelle stratégie.

Organisation pour l'implémentation de l'application:

a) Planning et répartition des tâches

Date début – Date fin	Tâche	Participant
08/01 – 15/01	UML + diagramme de séquence + répartition des tâches	Tout le monde
14/01 – 18/01	Etudes des différents design patterns pour le projet	Tout le monde
20/01 – 14/01	Interface	Tout le monde
25/01 – 31/01	Implémentation des indicateurs techniques	Pierre Romon Mathilde Le Moël
02/01 – 10/06	Implémentation de la classe transaction et transaction manager	Eloi Juszczak Tom Lecreux
02/01 – 04/06	Editeur de texte	Pierre Romon Mathilde Le Moel
05/06 – 08/06	Persistance des données – XML	Pierre Romon
05/06 – 10/06	Implémentation du mode pas à pas	Eloi Juszczak Tom Lecreux
10/06 – 14/06	Implémentation du mode automatique	Eloi Juszczak Tom Lecreux
12/06 – 15/06	Doxygen	Edmond Giraud
11/06 – 15/06	Rapport	Mathilde Le Moël

b) Organisation du travail et problèmes rencontrés :

Un des aspects les plus difficiles de ce projet a été d'organiser notre travail de groupe. Nous avons commencé par débattre longuement à propos de l'architecture de notre application et avons mis un certains temps avant de trouver un accord global quant à la modélisation UML à fournir. La réalisation de plusieurs diagramme de séquence au brouillon nous a finalement permis de conférer un certain dynamisme à notre modèle et ainsi de trouver un accord sur le modèle plus statique de l'UML.
Une fois cet accord trouvé, nous avons pu commencer la partie développement informatique du projet.

Afin de faciliter le travail, nous avons largement eu recours à la plateforme GitLab de l'utc. Nous avions pour habitude de travailler sur des branches séparées, chacune associée au développement d'un bloc du projet (mode pas à pas, chargement du CSV, mode automatique, etc).
Malgré l'utilisation de cet outil, il nous a parfois été difficile de coordonner toutes nos actions et nous avons dû retravailler sur une mise en commun afin de rendre certains codes compatibles.

Enfin un autre point important a été de faire évoluer notre modèle UML initial tout au long du projet. Bien que celui-ci ait été conservé dans sa globalité, nous nous sommes rendu compte de quelques petites défaillance dans sa définition initiale et il a fallu y apporter quelques modifications.

Conclusion :

La réalisation de ce projet nous a permis de mettre en application concrète différentes notions théoriques abordées tout au long de l'UV. Il nous a permis de nous familiariser avec différents design pattern à l'instar de Factory, Strategy ou encore Singleton et Iterator et nous a fait prendre conscience qu'il répondait à de réelles problématiques d'architecture et de conception orienté objet.

Au delà de l'aspect purement technique, ce projet a consolider notre expérience de travail en groupe et nous a confronté à différents problèmes auxquels nous avons dû trouver des solutions en autonomie, le travail de recherche sur la documentation QT s'étant révélé formateur et enrichissant.

Annexe:

UML: