



Rapport de Projet

Simulateur de Bourse

Sommaire

1	Introduction	2
2	Fonctionnalités principales	2
3	Architecture	4
3.1	Interface graphique	4
3.1.1	Diagramme de classes	4
3.1.2	Explication des classes	4
3.2	Parties Utilisateur et gestion financière	7
3.2.1	Diagramme de classes	7
3.2.2	Explication des classes	7
4	Dynamique du système	11
4.1	Collaboration entre objets	11
4.2	Circulation des données dans le système	11
4.3	Dépendances d'exécutions	11
5	Principaux choix de conception et réalisation	12
5.1	Importation et stockage des données	12
5.2	Mise à jour dynamique des données et interaction utilisateur	12
6	Méthodes agiles	13

Réalisé par :

Ralph Khairallah

Vadim Colin

Thomas Canisares–Morère

William Masson

Date : May 27, 2025

1 Introduction

Dans le cadre du projet long de Technologie Objet, nous avons choisi de développer une application intitulée **Simulateur de Bourse**. L'objectif principal de ce projet est de proposer un outil pédagogique permettant à tout type d'utilisateur, débutant ou initié d'apprendre à investir virtuellement en bourse.

À travers une interface simple mais complète, l'utilisateur pourra acheter et vendre des actions fictives, suivre l'évolution de son portefeuille, et observer en temps réel les effets de ses décisions. Ce simulateur vise à rendre plus accessibles les notions clés du monde de l'investissement : la gestion des risques, le suivi de performance, et la stratégie de placement dans différentes situations du marché boursier, comme un krach financier, une période de forte inflation, ou des événements exceptionnels comme la crise du COVID-19.

Ce rapport a pour but de présenter les différentes étapes de conception et de développement de l'application. Nous y détaillons l'architecture choisie, les choix techniques fais au fil du temps, les principales fonctionnalités mises en œuvre, ainsi que les difficultés rencontrées et les solutions apportées.

2 Fonctionnalités principales

Les principales user-stories de notre projet sont les suivantes :

Importer les données

L'objectif de cette user-story était de pouvoir importer efficacement des données de produits financiers sur de longues périodes de temps. Par exemple, pour notre application, on veut avoir accès aux prix d'ouverture et fermeture (prix au début et à la fin d'une journée) d'une action précise sur une période de temps donnée.

Cette user-story a été réalisée principalement durant les première et deuxième itérations et a été complétée.

Stocker les données

Une fois que les données ont été créées, il faut pouvoir les stocker de manière efficace afin d'y avoir facilement accès par la suite et pouvoir les réutiliser.

Nous avons travaillé sur cette user-story durant l'itération 2 mais avons surtout avancé lors de l'itération 3. Il y a encore quelques points que nous souhaiterions améliorer concernant cette user-story mais nous sommes néanmoins parvenus à stocker les données pour les réutiliser.

Afficher le cours d'un produit financier

Pour acheter un produit financier, l'utilisateur doit pouvoir être capable de suivre le cours de son prix au cours du temps. L'objectif de cet user-story est donc de mettre en place un affichage dynamique de ces courbes avec différents boutons permettant à l'utilisateur d'afficher ou d'enlever différents indicateurs financiers.

L'affichage du cours d'un produit financier a été réalisé durant l'itération 1 et les différents boutons ont été ajoutés durant l'itération 3.

Acheter et vendre un produit financier

L'utilisateur doit pouvoir acheter ou vendre le produit financier de son choix. Le solde de son portefeuille doit alors être modifié en fonction du prix du produit financier et ce produit doit être ajouté ou enlevé à la liste de ses possessions. Nous avons travaillé sur cette user-story lors de l'itération 3.

Accéder à n'importe quel produit financier

L'utilisateur doit pouvoir accéder à n'importe quel produit financier de son choix à travers une barre de recherche. L'application offre ainsi une grande flexibilité puisque les utilisateurs ne sont pas limités à seulement certains produits financiers choisis par les développeurs.

Cette étape a été complétée lors de l'itération 3. Néanmoins, même si cela fonctionne, certains points peuvent encore être améliorés.

Créer un compte et se reconnecter à une partie en cours

Un joueur doit avoir la possibilité de créer un compte afin de récupérer sa partie plus tard en se reconnectant. Les données concernant cet utilisateur doivent donc être stockées.

Nous avons avancé sur cette étape lors des itérations 2 et 3 mais nous n'avons pas réussi à la terminer. Nous avons créé les boutons qui permettent à un utilisateur de s'inscrire ou se connecter et nous arrivons à stocker les noms d'utilisateur et mots-de-passe mais le système ne fonctionne pas correctement.

Avancer dans le temps

L'application a pour but d'apprendre à l'utilisateur les rouages du monde de la bourse, pour cela, nous avons fait le choix de ne pas suivre son cours en temps réel. En effet, nous souhaitons plutôt proposer une version accélérée. Quand l'utilisateur choisit une nouvelle action, son prix est affiché sur une échelle de temps prédéfinie (par exemple de janvier 2025 à mars 2025). Ainsi, l'utilisateur peut choisir d'avancer dans le temps (par exemple jour par jour) pour voir tout de suite l'évolution du prix de l'action qu'il vient d'acheter.

Nous avons travaillé sur cette user-story lors des itérations 2 et 3. L'objectif final n'a pas été atteint, il est possible d'avancer dans le temps mais pas de se placer où l'on veut initialement : les données sont toujours affichées à partir de début 2025.

Se placer au cours d'un évènement historique

Toujours dans un but pédagogique, nous souhaitons que l'utilisateur puisse choisir dans quel intervalle de temps se placer pour qu'il puisse apprendre de grands événements historiques.

Nous n'avons pas eu le temps de commencer cette user-story.

3 Architecture

3.1 Interface graphique

3.1.1 Diagramme de classes

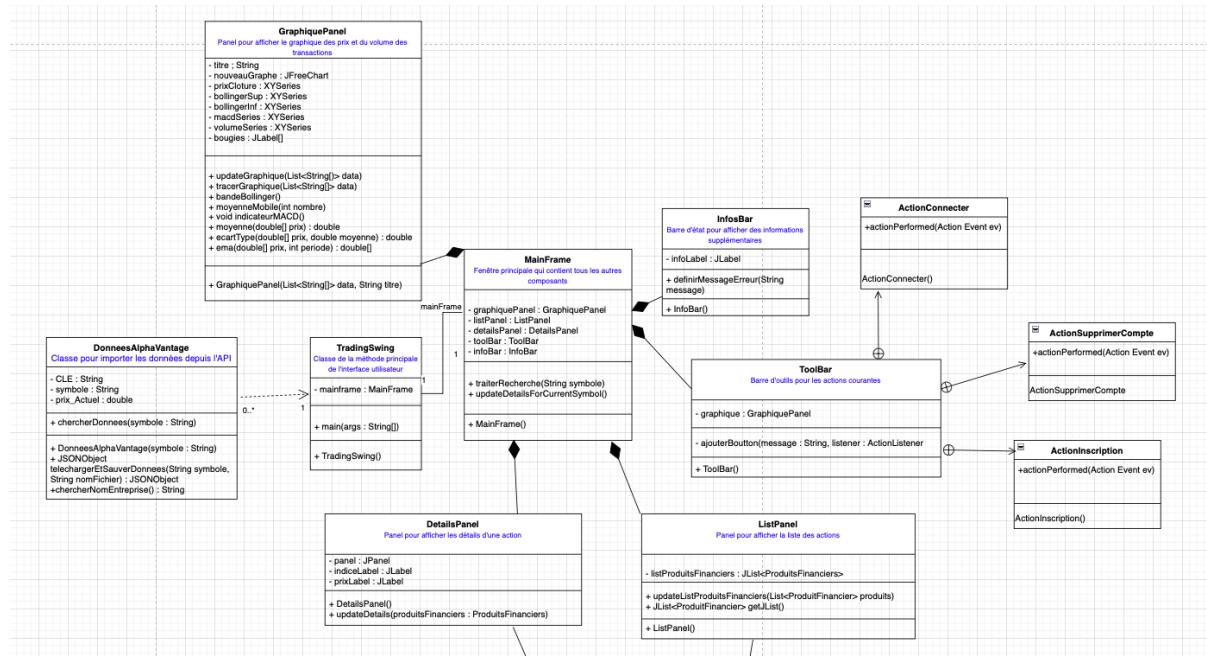


Figure 1: Partie du diagramme UML pour l'interface graphique

3.1.2 Explication des classes

- **GraphiquePanel** est un JPanel qui intègre deux graphiques générés avec JFreeChart

- le graphique principal affiche l'évolution du prix de clôture dans le temps ;
 - un graphique secondaire affiche les volumes.

Lorsqu'on appelle `setData(List<String[]> data)`, le graphique se met à jour avec les nouvelles données de l'action. La fenêtre de visualisation peut être agrandie avec le bouton “Avancer”.

La classe calcule aussi le **prix visible sur le graphique** (via `prixGraphe`) en fonction de la date affichée, ce qui est essentiel si on veut acheter une action au prix du graphique.

GraphiquePanel utilise un mécanisme d'Observer avec PropertyChangeSupport pour avertir les autres composants (comme MainFrame) lorsque le prix du graphique change.

- **DonneAlphaVantage** est une classe qui sert à interagir avec l'API Alpha Vantage. Elle est responsable de :

- `chercherDonne()` : méthode qui récupère les données journalières (open, high, low, close, volume) via API ou depuis un fichier JSON local (cache) ;
- `chercherNomEntreprise()` : méthode qui interroge l’API OVERVIEW pour obtenir le nom complet de l’entreprise (à partir de son symbole).

Les données sont stockées dans un fichier local pour limiter les appels à l’API (limités par minute). À chaque recherche, on vérifie si le fichier local est encore valide (en date), sinon on télécharge de nouveau les données.

- **MainFrame** est la classe principale de l’interface graphique. Elle hérite de `JFrame` et regroupe tous les composants (toolbar, graphique, panneau de gauche, panneau de droite, barre d’information, etc.).

Elle orchestre les interactions utilisateur et met à jour les composants en conséquence. Ses responsabilités incluent :

- `traiterRecherche(String symbole)` : méthode déclenchée lorsqu’on recherche une action. Elle :
 - * utilise `DonneAlphaVantage` pour charger les données ;
 - * met à jour le graphique via `graphiquePanel.setData(...)` ;
 - * extrait le prix du graphique (ajusté dans le temps) et met à jour `detailsPanel` ;
 - * ajoute l’action à la liste de gauche si elle n’y figure pas encore.
- `updateDetailsForCurrentSymbol()` : écoute les changements du graphique et met à jour dynamiquement les détails de l’action.

- **TradingSwing** est la classe contenant la **méthode main**. Elle initialise l’interface utilisateur en créant une instance de `MainFrame`, le point d’entrée graphique de l’application. Elle utilise `SwingUtilities.invokeLater` pour respecter les bonnes pratiques de Swing (exécution du code sur le thread graphique).

- **ToolBar** est une barre d’outils graphique qui regroupe tous les boutons d’actions disponibles dans l’application (recherche, indicateurs, connexion, inscription, etc.). Elle hérite de `JToolBar` et s’affiche en haut de la fenêtre.

Elle contient notamment :

- un champ de texte `champRecherche` dans lequel l’utilisateur peut saisir un symbole d’action à rechercher ;
- un bouton `Rechercher` qui déclenche la méthode `traiterRecherche()` de la classe `MainFrame` pour afficher les données de l’action demandée ;
- des boutons pour afficher les indicateurs graphiques comme les bandes de Bollinger ou la moyenne mobile ;
- les boutons d’inscription et de connexion, avec leur propre interface.

Le champ de recherche est également déclenché lorsqu’on tape sur la touche “Entrée” grâce à un `ActionListener`. Les erreurs (symbole invalide) ou succès (affichage graphique) sont relayés via la `InfoBar`.

➤ **InfoBar** est une simple barre textuelle située en bas de la fenêtre. Elle hérite de `JLabel` et permet d'afficher soit :

- des messages informatifs (ex. : “Données pour MCD”) via `mettreAJourInfo()`;
- des messages d'erreur avec style particulier (fond jaune, texte rouge, bordure) via `definirMessageErreur()`.

Elle sert de feedback utilisateur rapide à chaque interaction.

➤ **ListPanel** représente la colonne de gauche de l'interface utilisateur. Elle est organisée selon un `BorderLayout` et regroupe trois composants essentiels :

- une `JList<ProduitFinancier>` en haut, qui affiche les produits financiers que l'utilisateur a consultés via la recherche ;
- la vue du portefeuille utilisateur (`VuePortefeuille`) au centre ;
- le contrôleur de portefeuille (`ControlerPortefeuille`) en bas, pour acheter ou vendre les produits.

Cette structure reflète clairement l'architecture **MVC** :

- le modèle est représenté par le `Portefeuille` ;
- la vue est `VuePortefeuille` ;
- le contrôleur est `ControlerPortefeuille`.

La méthode `updateListProduitsFinanciers(...)` permet de recharger entièrement la liste d'actions visibles. Lorsqu'un produit est recherché et non encore listé, il est ajouté à ce composant.

➤ **DetailsPanel** est un panneau graphique situé à droite de l'interface utilisateur. Il fournit un **résumé visuel** des données principales liées à un produit financier sélectionné :

- **symbole** de l'action (ex. : “AAPL”, “TSLA”);
- **prix courant** du jour ;
- **prix affiché sur le graphique**, si celui-ci est ajusté dans le temps ;
- **variation en pourcentage** entre aujourd’hui et la veille.

Tous ces éléments sont mis à jour via la méthode `updateDetails()` lorsqu'on sélectionne un produit dans la liste ou lorsqu'on effectue une recherche.

`DetailsPanel` permet ainsi à l'utilisateur d'avoir une vue synthétique et immédiate de l'état actuel de l'action en question, sans devoir analyser le graphique.

3.2 Parties Utilisateur et gestion financière

3.2.1 Diagramme de classes

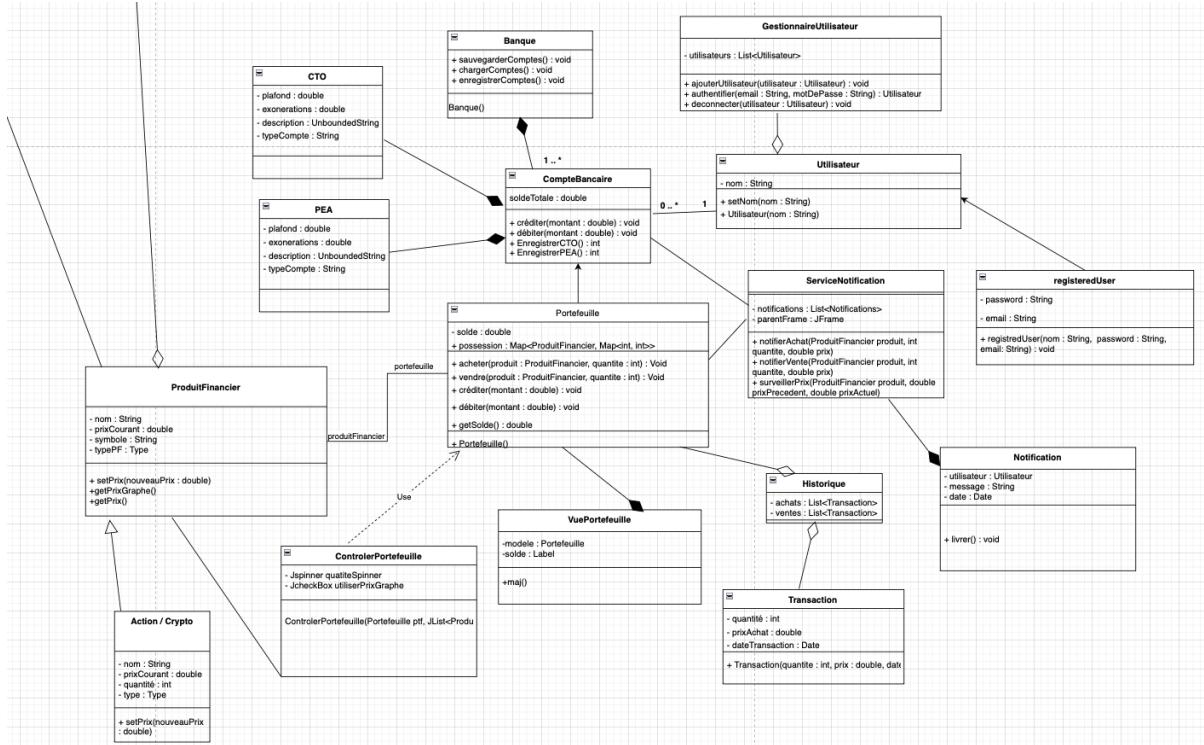


Figure 2: Partie du diagramme UML pour les parties utilisateur et gestion financière

3.2.2 Explication des classes

➤ **UserStorage** est une classe responsable de la **gestion en mémoire** des utilisateurs. Elle maintient deux listes : une liste générale **users** (tous les utilisateurs) et une liste **registeredUsers** (utilisateurs enregistrés avec email/mot de passe). Elle fournit des méthodes pour :

- ajouter un utilisateur ;
- supprimer un utilisateur (par pseudo ou email) ;
- récupérer les utilisateurs stockés ;
- afficher les utilisateurs.

Cette classe est utile pour centraliser l'enregistrement et la gestion des comptes en local avant leur éventuelle persistance.

- **User** est la classe de base représentant un utilisateur de l'application. Il possède un **pseudo** et un **CompteBancaire**. C'est à travers ce compte qu'il interagit avec les portefeuilles. Il constitue l'identité minimale d'un utilisateur.
- **RegisteredUser** hérite de **User** et ajoute des informations liées à l'authentification: **email** et **password**. Il est utilisé lors de la connexion et l'inscription. Cela permet de faire la distinction entre un utilisateur authentifié ou non.

- **Notification** représente un message destiné à l'utilisateur, daté et typé (achat, vente, alerte, etc.). Chaque notification contient un **message**, un **type**, et une **timestamp**. C'est une structure simple pour tracer les événements significatifs.
- **ServiceNotification** centralise la **gestion des notifications utilisateurs**. Il stocke une liste de notifications, et propose :
 - la création et l'affichage de messages pour chaque **achat ou vente** ;
 - l'envoi d'alertes en cas de **variation de prix importante** ;
 - l'affichage graphique des alertes via **JOptionPane**.

Il fonctionne en étroite collaboration avec **Portefeuille** et la **MainFrame**.

- **CompteBancaire** modélise un compte utilisateur contenant plusieurs **Portefeuille**, **CTO**, ou **PEA**. Il permet de :
 - **créditer ou débiter** le solde global ;
 - **enregistrer des portefeuilles**, en attribuant automatiquement une clé ;
 - gérer l'état des avoirs de l'utilisateur.

Il représente l'interface entre les produits financiers et les actions de l'utilisateur.

- **Transaction** décrit une opération d'achat ou de vente réalisée par l'utilisateur. Elle contient :
 - la **quantité** de titres échangés ;
 - le **prix d'achat unitaire** ;
 - la **date de la transaction**.

Elle est utilisée par le **Portefeuille** pour suivre l'historique des achats.

- **GestionnaireUtilisateur** est une classe utilitaire responsable de la **persistance des utilisateurs**. Elle utilise la bibliothèque **Gson** pour sauvegarder et charger les utilisateurs dans un fichier JSON (**utilisateur.json**). Elle contient aussi :

- une méthode pour connecter un utilisateur à partir d'un email et mot de passe ;
- une méthode pour enregistrer un utilisateur après son inscription.

Elle assure ainsi une passerelle entre le stockage local et l'état dynamique de l'application.

- **CTO** et **PEA** représentent deux types de comptes d'investissement, avec chacun ses plafonds fiscaux, ses exonérations et sa description réglementaire. Elles héritent toutes deux de **CompteBancaire**.
- **Banque** organise la création et l'ouverture de ces différents comptes.
- À l'intérieur d'un **CompteBancaire**, le **Portefeuille** prend en charge la détention et la transaction de produits financiers. Il conserve l'état actuel du portefeuille (la quantité de chaque titre) et fournit des méthodes pour acheter, vendre, créditer ou débiter.

➤ **Portefeuille** représente le cœur du modèle pour un utilisateur. Il gère à la fois :

- le **soldé en liquidités** disponible pour effectuer des transactions ;
- la **possession actuelle** des titres financiers (quantité détenue, prix d'achat, etc.) ;
- les **transactions réalisées**.

Il utilise une map `Map<ProduitFinancier, List<Transaction>` pour suivre les quantités achetées et les prix associés. Chaque clé est un produit (action, crypto, etc.), et la valeur est la liste des `Transaction` effectuées sur ce produit.

Il propose plusieurs méthodes principales :

- `acheter(ProduitFinancier, int, double)` : ajoute une transaction d'achat après vérification du solde ;
- `vendre(ProduitFinancier, int)` : retire une quantité détenue si disponible, et crédite le solde ;
- `crediter()` et `debiter()` : manipulent directement le solde ;
- `getQuantite(...)` : retourne le nombre total d'unités possédées d'un produit ;
- `getPossessions()` : donne accès à la map des titres détenus.

La classe hérite de `Observable`, ce qui permet à la `VuePortefeuille` de s'abonner au modèle. Toute modification (achat, vente, crédit, débit) déclenche automatiquement une mise à jour de la vue graphique.

En cas d'achat ou de vente, `Portefeuille` envoie aussi une notification au système via `ServiceNotification`, notamment pour avertir des changements de prix ou enregistrer une alerte importante.

➤ **ProduitFinancier** est une classe **abstraite** qui représente tout actif pouvant être détenu dans un portefeuille : une action, une crypto-monnaie, un ETF, etc.

Elle définit les propriétés et comportements communs à tous les produits :

- `getNom()` et `getSymbole()` : fournissent respectivement le nom de l'actif et son code boursier ;
- `getPrixCourant()` : retourne le prix actuel de l'actif ;
- `getPrixGraphe()` : retourne le prix au moment affiché dans le graphique ;
- `setPrixCourant(...)` et `setPrixGraphe(...)` : permettent de mettre à jour dynamiquement les prix affichés.

Cette classe est conçue pour être étendue. Par exemple :

- `ActionGenerique` est une sous-classe concrète utilisée pour les actions recherchées dynamiquement par l'utilisateur ;
- `CryptoMonnaie` ou d'autres produits peuvent être implémentés ultérieurement avec la même structure.

Grâce à cette abstraction, l’application peut manipuler tous les types de titres de manière uniforme, que ce soit pour les afficher, les acheter, ou les analyser graphiquement.

➤ VuePortefeuille

La classe `VuePortefeuille` représente la **vue graphique du portefeuille de l’utilisateur**. Elle hérite de `JPanel` et affiche deux éléments principaux :

- un `JLabel` qui indique le **solde actuel** de l’utilisateur ;
- un `JTable` qui liste les produits financiers détenus (nom, quantité, prix d’achat).

`VuePortefeuille` observe le modèle `Portefeuille`, qui hérite de `Observable`. Lorsqu’une transaction est effectuée (achat ou vente), le modèle notifie automatiquement la vue, qui appelle la méthode `maj()` pour mettre à jour l’affichage.

`maj()` parcourt la map des possessions du portefeuille et recharge entièrement le tableau. Chaque ligne affiche :

- le nom du produit (`ProduitFinancier.getNom()`) ;
- la quantité possédée (extrait depuis les `Transaction`) ;
- le prix d’achat.

Ainsi, `VuePortefeuille` constitue une interface visuelle fidèle et synchronisée avec les données du modèle.

➤ ControlerPortefeuille

La classe `ControlerPortefeuille` est un composant graphique qui sert de **contrôleur d’interaction pour le portefeuille**. Elle permet à l’utilisateur d’effectuer des opérations d’achat et de vente sur les produits financiers sélectionnés dans l’interface.

Elle contient :

- un `JSpinner` pour choisir la quantité à acheter ou vendre ;
- une `JCheckBox` permettant de sélectionner l’utilisation du **prix du graphique** (plutôt que le prix courant) ;
- deux `JButton` : “Acheter” et “Vendre”.

Lorsqu’un des boutons est cliqué :

1. le produit sélectionné dans la `JList<ProduitFinancier>` est récupéré ;
2. la quantité est lue dans le `JSpinner` ;
3. le prix utilisé dépend de l’état de la `JCheckBox` (`getPrixGraphe()` ou `getPrixCourant()`) ;
4. la méthode `ptf.acheter(...)` ou `ptf.vendre(...)` est appelée avec les bons paramètres ;
5. si la transaction est acceptée, la vue du portefeuille est automatiquement rafraîchie.

Des boîtes de dialogue sont affichées en cas d'erreur (produit non sélectionné, fonds insuffisants, ou quantité invalide).

ControlerPortefeuille et **VuePortefeuille** sont tous deux intégrés dans le **ListPanel**, assurant une séparation claire entre la logique métier (modèle), les interactions (contrôleur) et l'affichage (vue).

4 Dynamique du système

4.1 Collaboration entre objets

Lorsqu'un utilisateur souhaite acheter un produit financier, l'interaction débute par l'interface utilisateur, via **ControlerPortefeuille**, où l'utilisateur clique sur un bouton dédié. Cet événement est capté par le **MainFrame**, qui agit comme point d'entrée et de coordination de l'application.

Le **MainFrame** fait appel au portefeuille associé à l'utilisateur, représenté par la classe **Portefeuille**, pour initier l'opération d'achat. Ce dernier, à son tour, consulte le **CompteBancaire** pour vérifier que les fonds sont suffisants, puis procède au débit du montant requis.

Ensuite, le portefeuille met à jour la possession de produits financiers, et crée une instance de la classe **Transaction** pour archiver l'achat. Cette transaction est ensuite ajoutée à l'**Historique** de l'utilisateur. En parallèle, une notification est générée via le **ServiceNotification** et adressée à l'utilisateur pour l'informer de la réussite de l'opération.

Enfin, le **MainFrame**, à travers l'objet **InfoBar**, affiche un message de confirmation visuel.

4.2 Circulation des données dans le système

Les données circulent dans le simulateur de manière structurée. Les cours des produits financiers, par exemple, sont récupérés au démarrage et en appuyant sur le bouton "Avancer" de la **ToolBar**, par la classe **DonnéesAlphaVantage**, qui se connecte à l'API externe **AlphaVantage**.

Ces données sont ensuite stockées dans les objets **ProduitFinancier**, puis exploitées par les interfaces comme **GraphiquePanel** ou **ListPanel** pour les représenter graphiquement ou sous forme de liste.

Lorsqu'un utilisateur effectue un achat, les données saisies dans l'interface deviennent des transactions, et servent à ajuster l'état du portefeuille et du compte bancaire. Elles transitent aussi vers le service de notification pour informer l'utilisateur, et vers les composants d'affichage comme la **InfoBar** pour mise à jour en temps réel.

De plus, les indicateurs boursiers tels que les moyennes mobiles ou les bandes de Bollinger sont recalculés à partir des données de prix stockées dans les séries temporelles (XYSeries) du **GraphiquePanel**, assurant un affichage dynamique des variations de marché.

4.3 Dépendances d'exécutions

L'interface graphique, notamment les classes **DetailsPanel** et **ListPanel**, dépendent des composants métiers tels que le Portefeuille, le **CompteBancaire**, ou encore le **Ser-**

viceNotification. Ces derniers sont sollicités pour exécuter les opérations courantes, comme acheter ou vendre.

Les modules métiers, de leur côté, dépendent de l'accès aux données de marché fournies par **DonnéesAlphaVantage**, qui constitue une dépendance externe liée à l'API.

Enfin, le mécanisme de notification et d'affichage repose sur une coordination entre le corps métier et l'interface, à travers l'appel à ServiceNotification et à la mise à jour de InfoBar.

5 Principaux choix de conception et réalisation

5.1 Importation et stockage des données

Pour mettre en place notre application, nous devions avoir accès à un grand nombre de données de produits financiers, en particulier parce que nous souhaitions que l'utilisateur puisse accéder à n'importe quel produit de son choix. Pour cela, nous avons utilisé une API qui nous permet de récupérer les données d'une action au format JSON à partir d'une clé et du symbole de cette action. Nous avons ensuite écrit une méthode qui traite ces données et les met sous forme d'un tableau ce qui permet de les utiliser efficacement. Ainsi, l'utilisateur n'a qu'à taper le nom d'une action dans la barre de recherche et il peut observer son cours, en acheter une certaine quantité.

Cependant, dans notre solution initiale, les données n'étaient pas directement stockées dans un fichier JSON une fois qu'elles avaient été importées. En effet, nous devions effectuer une requête vers l'API à chaque fois que nous devions redémarrer l'application ou afficher un nouveau graphique.

Cette conception posait deux problèmes :

- L'application était assez lente puisque effectuer une requête vers l'API prend un certain temps.
- Nous ne disposons que de 20 requêtes API si bien qu'assez rapidement l'application était bloquée, nous ne pouvions plus afficher d'actions.

C'est pourquoi nous avons fait le choix de stocker directement les données d'une action dans un fichier JSON quand on les importe.

Ainsi, quand un utilisateur essaye d'importer une nouvelle action, l'application vérifie d'abord s'il n'existe pas un fichier JSON qui correspond à cette action pour ne pas avoir à faire une requête vers l'API.

5.2 Mise à jour dynamique des données et interaction utilisateur

L'une des fonctionnalités essentielles de notre application est sa capacité à réagir dynamiquement aux actions de l'utilisateur. Grâce à la **ToolBar**, l'utilisateur peut rechercher une action en tapant son symbole et en appuyant sur "Entrée" ou sur le bouton **Rechercher**, ce qui déclenche automatiquement la récupération ou le chargement local des données correspondantes.

Chaque action recherchée est ajoutée à une liste interactive affichée à gauche de l'écran. En cliquant sur une action, le graphique central se met à jour automatiquement, et les informations (symbole, prix courant, prix du graphique, variation) sont rafraîchies dans la section droite.

De plus, lors de l'achat d'un produit, l'utilisateur peut choisir d'utiliser le prix courant ou celui affiché sur le graphique (à un instant donné simulé grâce au bouton **Avancer**). Cette précision rend le simulateur plus réaliste, car elle permet d'acheter une action à un instant spécifique du temps simulé.

6 Méthodes agiles

Au commencement du projet, nous avons choisi de programmer plusieurs réunions pour affiner les objectifs du projet et commencer à écrire l'UML.

Dès lors, nous avons décidé de séparer le projet en deux grandes parties, l'une concernant l'IHM et l'autre le corps métiers qui englobe la gestion des utilisateurs et des portefeuilles.

Chaque personne a pu choisir de se spécialiser dans une des deux parties selon ses préférences. L'objectif étant de ne pas s'éparpiller et d'éviter de se marcher dessus.

Pour se répartir le travail de manière plus précise, nous avons décidé d'utiliser Trello. Nous avons créé un grand nombre de user-stories sur lesquels chacun pouvait se positionner à sa guise.

Le principal moyen de communication que nous avons utilisé est Whatsapp. Chacun essayait d'annoncer sur quelles classes il comptait travailler avant de les modifier pour éviter les conflits sur le git. Nous avons également essayé de faire un retour sur le groupe Whatsapp à chaque fois que nous avancions pour tenir les autres membres informés de l'évolution du projet.

Néanmoins nous nous sommes sommes appréciés au cours du projet que la manière de travailler qui nous réussissait le plus était en présentiel dans les salles de l'école, sur des ordinateurs côte-à-côte. En effet, cela nous permettait d'échanger facilement et d'avancer bien plus rapidement que chacun de notre côté. Nous avons donc programmé des sessions de une ou deux heures pendant lesquelles chacun pouvait venir coder en groupe.

Cette solution s'est avérée efficace puisque les classes étaient très reliées entre elles si bien que nous avions du mal à travailler en même temps si nous n'étions pas côte-à-côte.