



Département Génie Industriel et Mathématiques Appliqués

---

# Rapport de stage

---

**Sujet :** Revue de l'architecture du pricing des options FX



**Établissement :** École Nationale Supérieure des Mines de Nancy

**Entreprise d'accueil :** Murex

*Auteur :*  
M. BERRADA Hassan

*Tuteur Entreprise :*  
M. AMRAOUI Mohamed

*Tuteur École :*  
M. VILLEMONAIS Denis

6 mars 2023 - 4 août 2023

# Sommaire

<b>Introduction</b>	<b>3</b>
<b>1 Présentation de l'entreprise</b>	<b>5</b>
1.1 Historique et évolution . . . . .	5
1.2 Mission de l'entreprise . . . . .	5
1.3 Positionnement sur le marché . . . . .	6
1.4 Structure de l'entreprise . . . . .	7
1.5 Présentation de l'équipe . . . . .	8
<b>2 Contexte et objectif du stage</b>	<b>10</b>
2.1 Les Tests en Software Development . . . . .	10
2.2 Méthodologie d'intégration et de déploiement continu . . . . .	11
2.3 L'application MX.3 . . . . .	15
2.4 Présentation de la problématique . . . . .	15
2.5 Cahier des charges et objectif . . . . .	17
<b>3 Développement de ma mission</b>	<b>20</b>
3.1 Options sur devises . . . . .	20
3.2 Sensibilités des options . . . . .	21
3.3 Présentation de l'architecture actuelle du code de Pricing . . . . .	22
3.4 Organisation de mon travail et gestion de mon projet . . . . .	23
3.5 Premières missions de refactoring de code . . . . .	24
3.6 Écriture des tests unitaires offline . . . . .	27
3.7 Mise en place de l'extension TestMate . . . . .	28
3.8 Deuxièmes missions de refactoring de code . . . . .	29
<b>4 Retour sur mon expérience</b>	<b>31</b>
4.1 Retour sur mes objectifs . . . . .	31
4.2 Mission de développeur . . . . .	31
4.3 Écueils que j'ai rencontré . . . . .	32
<b>Conclusion</b>	<b>32</b>
<b>Bibliographie</b>	<b>33</b>

<b>Glossaire</b>	<b>34</b>
<b>A Annexes</b>	<b>36</b>
A.1 Tâches attribuées . . . . .	36
A.2 Comparaison de MX et de MxFTest . . . . .	38
A.3 Script d'utilisation de Testmate . . . . .	39
A.4 Interface de TestMate . . . . .	48
A.5 Couverture des tests . . . . .	49

# Remerciements

Avant tout, je tiens à exprimer ma gratitude à l'ensemble de l'équipe fxo pour son accueil chaleureux et qui m'a permis de grandement apprécier le moment partagé et les échanges que j'ai pu avoir lors de mon stage.

Ensuite, mes remerciements vont à mon tuteur de stage M. Mohamed Amraoui, qui m'a guidé sur l'ensemble de mon stage et qui a su être très présent pour mener à bien ma mission.

Enfin, À M. Omar Alami Ouali, M. Elom Foli-Bele et Alain Rakotomamonjy qui ont été respectivement très présents pour me former sur l'ensemble de mon stage, en me donnant les outils nécessaires pour évoluer dans ma mission.

# Introduction

Le stage de fin d'études ingénieur a pour objectif d'appliquer les compétences acquises au cours de la formation d'ingénieur généraliste à Mines Nancy. Les compétences à mettre en valeur lors de ce stage sont de trois natures distinctes. Premièrement, les compétences techniques, qui sont centrales pour mener à bien la mission. Ensuite, les compétences de communication et de travail d'équipe, qui permettront une collaboration efficace au sein de l'équipe mais également de véhiculer ses idées et être proactif du développement. Enfin, les compétences en gestion de projet, qui seront essentielles pour atteindre les objectifs fixés en termes de délais, coût et qualité.

Ayant effectué ma deuxième année au département informatique et ma troisième année dans le département ingénierie mathématiques dans le parcours mathématiques financières, j'ai cherché un stage qui puisse couvrir ces deux domaines afin de pouvoir mettre en pratique les compétences acquises ces deux années. Mon semestre d'échange à l'Université Technique de Munich m'avait également permis de prendre des cours dans ces deux domaines. C'est donc naturellement que je me suis tourné vers Murex, une entreprise de développement de logiciel destinés aux traders pour un stage de développement au sein de l'équipe qui développe le module de trading pour les options sur devises.

La mission de mon stage consistera à développer la testabilité et les tests du module de pricing de dérivés sur les options. Actuellement, ce module n'est pas adapté aux tests unitaires et mon objectif sera de développer les tests de ce module ainsi que d'améliorer sa testabilité. Ainsi, le stage nécessitera une bonne maîtrise de la programmation C++ mais également une bonne connaissance du fonctionnement des produits dérivés sur les devises.

# Chapitre 1

## Présentation de l'entreprise

### 1.1 Historique et évolution

Murex, fondée en 1986 par Salim Edde et Laurent Néel, est née en pleine période de décloisonnement et déréglementation des marchés financiers. Durant les années 1980 et 1990, le secteur financier a connu une libéralisation massive, notamment en Europe avec la création du Marché à Terme International de France (MATIF) en 1986. Cet environnement a conduit à une multiplication du volume des transactions financières et à une complexification des produits échangés. Par conséquent, une numérisation du processus d'échange ainsi que de l'aide à la décision pour le trading est devenue nécessaire.

À l'heure actuelle, en 2023, Murex compte plus de 2700 employés répartis sur 19 pays. Les opérations de la société sont centralisées dans ses deux principaux bureaux situés à Paris et à Beyrouth. Néanmoins, la présence mondiale de Murex témoigne de son rôle significatif dans l'industrie financière internationale, et compte à ce jour plus de 60 000 utilisateurs. En effet, la plateforme Murex est responsable de 25% des échanges se produisant au sein du marché des devises, soit un flux journalier de plus de 2 000 milliards de dollars traité par Murex.

### 1.2 Mission de l'entreprise

Murex se concentre sur le développement et la maintenance d'un logiciel singulier, MX.3, qui est commercialisé auprès des clients sous forme de modules. Ils peuvent choisir et acheter certains de ces modules en fonction de leurs besoins. Les modules proposés par Murex peuvent être classifiés en trois principales catégories d'activités : le front office, le back office et la gestion de trésorerie. Le front office est dédié aux opérations de marché, comme l'exécution des ordres de vente et d'achat d'instruments financiers. Le back office est centré sur l'administration et le support des activités de trading, comme la gestion de portefeuilles, la conformité réglementaire et la gestion des risques. En dernier lieu, et plus récemment, Murex développe la gestion de la trésorerie et des actifs afin d'élargir son champs d'action.

### 1.3 Positionnement sur le marché

Le business model de Murex repose sur trois stratégies distinctes. La première, qui constitue la principale source de revenus de l'entreprise (72% du chiffre d'affaires), consiste en la vente de licences du logiciel avec des frais de maintenance associés. Au fur et à mesure que Murex se développait, l'entreprise a ensuite permis la location de son logiciel. Cette location représente 22% du chiffre d'affaires, mais son horizon temporel limité (5 ans contre 10 ans pour la vente) tend à rendre cette option moins efficace pour la fidélisation des clients. C'est pourquoi cette option est uniquement disponible pour ses contrats les plus conséquents. Plus récemment, Murex a adopté un modèle de Software as a Service (SaaS), où l'accès au logiciel est donnée par l'intermédiaire d'une machine distante. Ce dernier modèle, bien que ne constituant que 6% du chiffre d'affaires, se développe rapidement et devrait, à terme, attirer un plus grand nombre de clients à travers sa facilité d'utilisation.

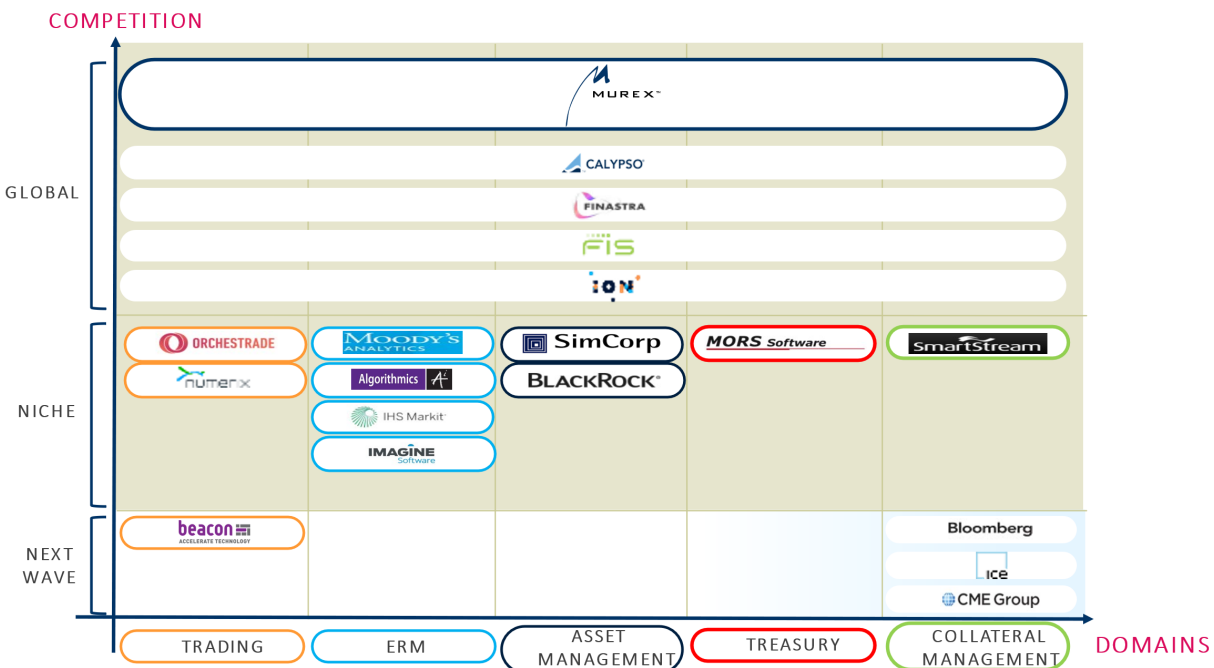


FIGURE 1.1 – Domaines d'activité des éditeurs de logiciel pour la finance

Murex occupe une position singulière sur le marché des éditeurs de logiciels pour la finance. En effet, cette entreprise est une des figures majeures de ce secteur de par son ancienneté. Murex a su garder un avantage par rapport à sa compétition en diversifiant son activité. En effet, Murex est l'un des rares éditeurs à proposer des solutions couvrant l'intégralité du marché financier, soit, le front-office, le back-office et la gestion des risques,

des actifs et de la trésorerie.

Murex diversifie également ses clients et types de clients. Les clients les plus gros de Murex ne représentent pas plus de 6 % du chiffre d'affaire. L'entreprise se positionne dans le buy side<sup>†</sup> ou le sell side<sup>†</sup>. Ses clients peuvent donc être tout autant des asset manager, des hedge fund ou des banques d'investissement régionales ou globales.



FIGURE 1.2 – Types de clients chez Murex

## 1.4 Structure de l'entreprise

L'entreprise adopte la méthode SAFe (Scaled Agile Framework) afin d'organiser sa structure. Comme son nom l'indique, SAFe est une méthode qui contient des principes généraux permettant d'organiser une entreprise autour de la méthode agile. Elle donne un ensemble de principes afin d'orchestrer de multiples équipes qui travaillent en concert.

C'est pourquoi Murex est divisée en trains. Un train constitue d'un ensemble d'équipes travaillant sur des fonctionnalités bien spécifiques le plus indépendamment possible. Un train est à son tour constitué de plusieurs équipes qui travaillent sur des modules différents. Cela permet de répondre à une des exigences de SAFe qui est de décentraliser la prise de décision.

Par exemple, le train dans lequel je me trouve est le train *instruments*. Ce train constitue l'ensemble des équipes qui travaillent dans le développement des outils pour les actifs financier (obligations, actions, devises, options sur devises, etc...). Mon équipe par exemple travaille sur les options sur devises.



De plus chaque équipe possède une structure particulière. Dans notre train il est commun d'avoir une équipe constituée de développeurs (dev), de consultants fonctionnels (PES) et de contrôleurs de qualité ou quality assurance (QA). Dans d'autres trains il est possible d'avoir uniquement des consultants clients (Client Services) qui sont en contact direct avec les clients pour répondre à leurs besoins. Lorsque ces besoins consistent en la gestion d'un bug, les CS communiquent ce besoin au PES qui ont pour tâche d'isoler et spécifier ce problème. Il est ensuite communiqué aux développeurs afin d'être corrigé dans la version du client.

## 1.5 Présentation de l'équipe

L'équipe FXO comme son nom l'indique est l'équipe qui gère les options sur devise. Actuellement, elle est constituée de 7 développeurs dont 5 développeurs senior, de 9 consultants PES (Product Evolution Services), dont 7 seniors et de 2 QA (Quality Assurance). L'équipe est donc une équipe relativement grande. Elle s'organise à l'aide du framework scrum comme la plupart des équipes de Murex.

Ce framework permet d'organiser l'équipe de manière efficace et dynamique, en adaptant régulièrement et rapidement le travail à faire. L'équipe FXO, comme l'ensemble des équipes de développement de Murex, s'organise en PI (Program Increment). Un PI est un bloc temporel spécifique, typiquement de trois mois, pendant lequel un ensemble prévisible de travail doit être accompli. Le PI est subdivisé en sprints individuels, chacun de deux semaines. Avant le début de chaque PI, un PI Planning est réalisé où les objectifs pour le prochain PI sont définis et le travail nécessaire pour atteindre ces objectifs est organisé.

En termes de développement logiciel, chaque sprint permet à l'équipe de travailler sur des sections spécifiques du code, qui sont ensuite consolidées à travers un archivage du code pour préparation à leur intégration dans la version principale du logiciel. En regroupant les sprints dans un PI, l'équipe a un cadre temporel plus grand pour travailler vers des objectifs plus complexes ou à long terme (parfois nommés *epics*<sup>†</sup>), tout en gardant le rythme et la flexibilité offerts par les sprints individuels.

De plus, certains rôles sont attribués à chaque membres de l'équipe comme le rôle de scrum master. Il orchestre l'application de la méthodologie scrum et s'assure du bon fonctionnement de celle-ci. Concrètement, celui-ci organise les daily meetings et s'assure du bon avancement des tâches (appelées *stories*<sup>†</sup>).

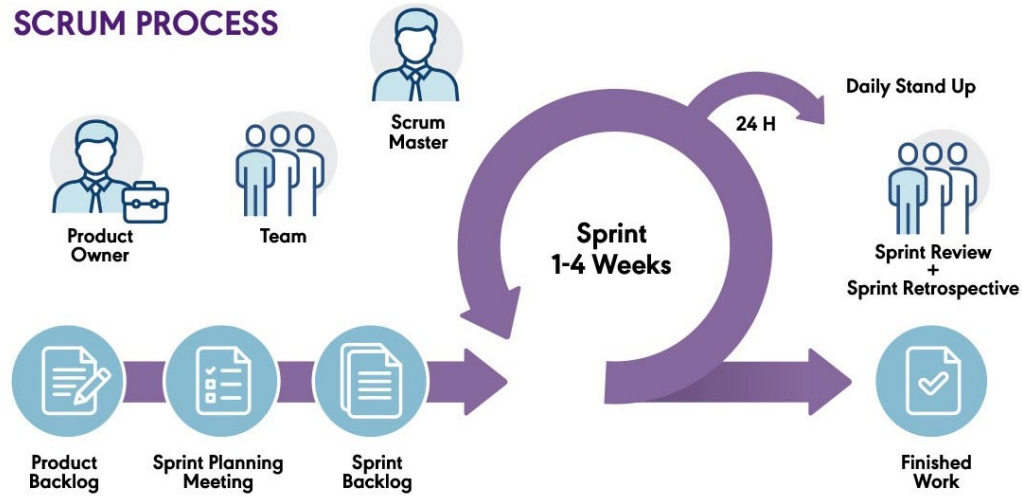


FIGURE 1.3 – Illustration du processus Scrum

# Chapitre 2

## Contexte et objectif du stage

### 2.1 Les Tests en Software Development

La garantie de la qualité d'un logiciel est primordiale dans son développement. Une façon de l'assurer est à travers une stratégie de tests robuste, souvent illustrée par la pyramide des tests.

La base de la pyramide des tests est constituée de tests unitaires. Ceux-ci sont les plus nombreux et visent à tester les plus petites unités de code, généralement des fonctions ou des méthodes, pour s'assurer qu'elles se comportent comme prévu. Les tests unitaires sont rapides à exécuter et permettent une détection précoce des anomalies à l'échelle la plus basique du code.

Au milieu de la pyramide se trouvent les tests d'intégration. Ces tests vérifient comment une certaine unité de code ou module interagit avec le reste du logiciel. Cela peut inclure le test de la communication entre différentes parties du code, entre différentes bases de données, ou entre le système et des services tiers. Les tests d'intégration permettent de détecter les erreurs qui peuvent survenir lorsque des unités de code différentes sont combinées.

Enfin, au sommet de la pyramide, nous avons les tests End to End. Ils représentent une proportion plus petite du volume total de tests, mais sont cruciaux pour garantir que l'ensemble du système fonctionne correctement dans un environnement semblable à celui des clients. Les tests End to End vérifient le flux complet d'une application, de bout en bout, pour s'assurer que tout le système fonctionne ensemble comme prévu en simulant l'expérience utilisateur le plus fidèlement possible.

En résumé, chaque niveau de la pyramide de tests joue un rôle important et indépendant dans la garantie de la qualité du logiciel. Les tests unitaires assurent la fiabilité du code à bas niveau, les tests d'intégration garantissent que les différentes parties du code interagissent correctement, et les tests End to End confirment que le système dans son ensemble fonctionne comme il se doit dans un environnement similaire à celui de production.

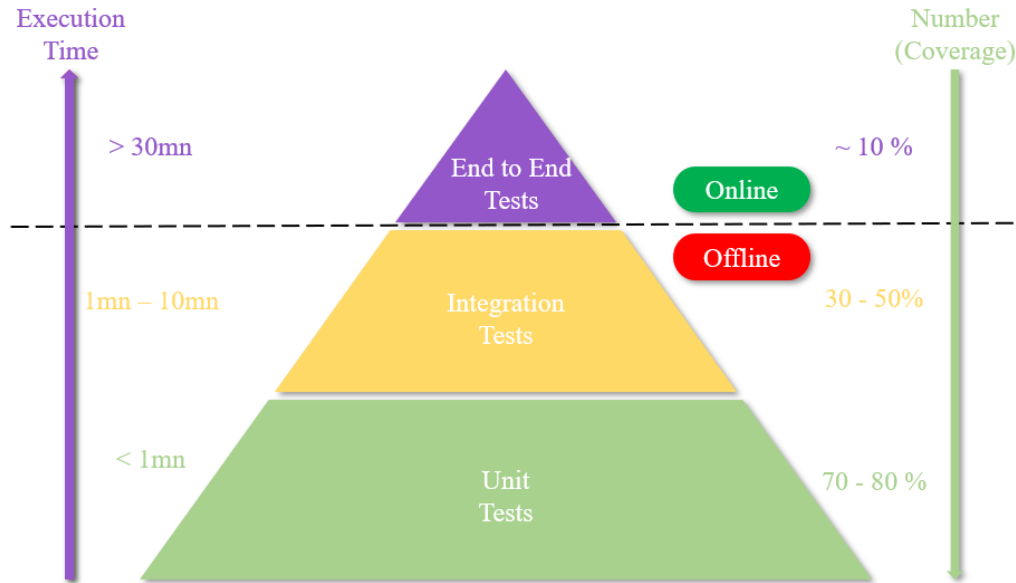


FIGURE 2.1 – Pyramide des tests

Idéalement, comme le représente la pyramide de tests, les tests unitaires qui sont les plus rapides à exécuter devraient être les plus nombreux. Ensuite viennent les tests d'intégration qui doivent être moins nombreux que les tests unitaires mais doivent capturer le fonctionnement d'un module dans le reste de l'application. Enfin, les tests End to End doivent être moins nombreux car ceux-ci sont bien plus chronophages et ne doivent capturer que les cas d'utilisations les plus complexes.

Une distinction importante dans les tests est le caractère online/offline. Les tests online impliquent une connexion active à un système ou une ressource, comme une base de données ou un serveur web, pour valider l'interaction du système avec ces ressources et sont en général des tests End to End. En revanche, les tests offline sont effectués sans aucune connexion à une ressource externe. Ils permettent de tester la logique interne du système et peuvent souvent être exécutés plus rapidement que les tests online.

## 2.2 Méthodologie d'intégration et de déploiement continu

### 2.2.1 Outils pour l'intégration et le déploiement continu

Un des principes de la programmation agile est l'intégration et le déploiement continu. Murex adopte une organisation d'intégration et de déploiement continu assez avancée. On parle souvent de CI/CD (Continuous Integration/ Continuous Deployment). Pour cela, un

ensemble d'outils sont nécessaires :

- Un outil de communication
- Un outil de gestion de projet agile.
- Un outil de gestion de code et de versions.
- Un outil de gestion de tests automatisé.
- Un outil de gestion d'intégration continue.
- Un outil de déploiement continu

Pour ces différents types de besoins, Murex peut utiliser un ou plusieurs outils qui sont pour la plupart des outils de pointe dans l'industrie. Par exemple, l'outil de gestion de projet agile est jira qui est un outil externe très performant. Cet outil permet d'orchestrer la méthode agile au niveau global de l'entreprise. C'est cet outil qui permet par exemple de créer des stories qui sont la brique de base d'un changement dans le code. Il permet également de suivre les stories en cours et de les classer par PI et sprint ou epic.

Quant à la gestion des versions, Murex utilise Perforce, un outil externe puissant qui assure le contrôle des différentes versions du code. Un outil analogue communément utilisé pour ce besoin est git. La différence entre ces deux outils est la gestion décentralisée pour git contre la gestion centralisée pour Perforce. Cette différence provient de la manière dont le stockage des versions est géré. Contrairement à Git où le code et son historique sont stockés localement, Perforce conserve le code de manière centralisée avant de le distribuer aux utilisateurs. Perforce est donc particulièrement approprié pour des projets d'envergure, comme Murex, nécessitant une intégration rapide des changements dans une vaste base de code. Chaque intégration de code devient immédiatement la référence pour toute l'équipe..

Enfin, Murex utilise plusieurs outils pour la gestion des tests automatisés et pour l'intégration continue. En effet, l'entreprise va utiliser Jenkins pour le lancement de pipelines et routines pour les tests automatiques du code mais également des pipelines d'intégration continue du code. Jenkins permet de monitorer et d'exécuter ces programmes de façon très efficace. D'autres part, Murex va également utiliser un outil de tests automatisé pour ses tests End to End qui est GQAF. Il permet de lancer des tests automatiques sur des machines distantes et d'observer les résultats de ces machines en les comparant aux résultats attendus. Pour l'intégration dans la version de build, Murex utilisera un outil interne qui est l'integration-bot et qui permet de manière similaire de lancer des pipelines jenkins et une série de tests End to End.

Toujours dans le cadre de la méthode de développement agile, d'autres outils sont fortement recommandés afin faciliter la communication et le développement.

- Un outil de documentation
- Un outil de révision des modifications du code
- Un outil de visualisation de la couverture des tests
- Un outil de suivi et gestion des bugs

Pour la documentation, Murex va utiliser plusieurs outils qui ont une fonction bien précise comme une documentation générale avec un Wiki, une documentation du code avec opengrok permettant la recherche textuelle et doxygen permettant de générer une documentation fonctionnelle du code se basant sur la structure des objets. Pendant le développement, un développeur peut notifier l'équipe de ses modifications à l'aide de Helix Swarm qui permet de faire de la révision des modifications.

### 2.2.2 Processus d'intégration du code

Murex organise son code à l'aide de versions. Il existe principalement trois types de versions. Premièrement la version d'équipe permet à une équipe de modifier le même code lors d'un sprint. Lorsqu'une personne modifie son code, celui-ci doit être intégré dans la version d'équipe en premier lieu. Un certain nombre de critères doivent être alors vérifiés sur le code qui permettent d'assurer le niveau de qualité 0 du code (ou Quality Level 0, abrégé QL0) afin de l'intégrer dans la version d'équipe. Ce niveau est attribué au code s'il passe l'ensemble des tests End to End (TPK<sup>†</sup> ou Test Package) de l'équipe (~20 TPKs) ainsi que l'ensemble de tout les tests unitaires de Murex. D'autres TPKs (~60 TPKs) sont également lancés après que le code soit intégré dans la version d'équipe désignés comme SafetyNet contenant non seulement des Test End to End mais également des tests de performance et de mémoire.

Ensuite, il est possible de l'intégrer dans la version de build qui est une version commune à l'ensemble des équipes. Cette version fait foi comme la version de développement de l'application et sert de point de référence à l'ensemble des équipes. Afin d'être intégré dans la version build, le code doit atteindre le niveau de qualité 1 (QL1) qui est attribué également par un ensemble de TPKs étant des TPKs qui assurent le fonctionnement de base de l'application MX.3. Chaque 2 jours, la version de build est archivée afin de passer des tests exhaustifs de l'application permettant d'assurer le niveau de qualité 3 (QL3) comme l'ensemble des TPKs (~900 TPKs et des tests unitaires de Murex ainsi que des vérifications de la qualité du code.

Ce sont ces versions d'archives qui sont ensuite utilisées en début de chaque sprint comme point de référence pour créer les versions d'équipe. Après chaque PI, soit trois mois, une version d'archive est choisie afin de représenter une version de déploiement (ou release). Les clients ont alors le choix de mettre à jour ou non leur logiciel Murex sur cette nouvelle version sachant que les clients acceptent des mises à jour quelques années après la dernière (entre 3 et 4 ans).

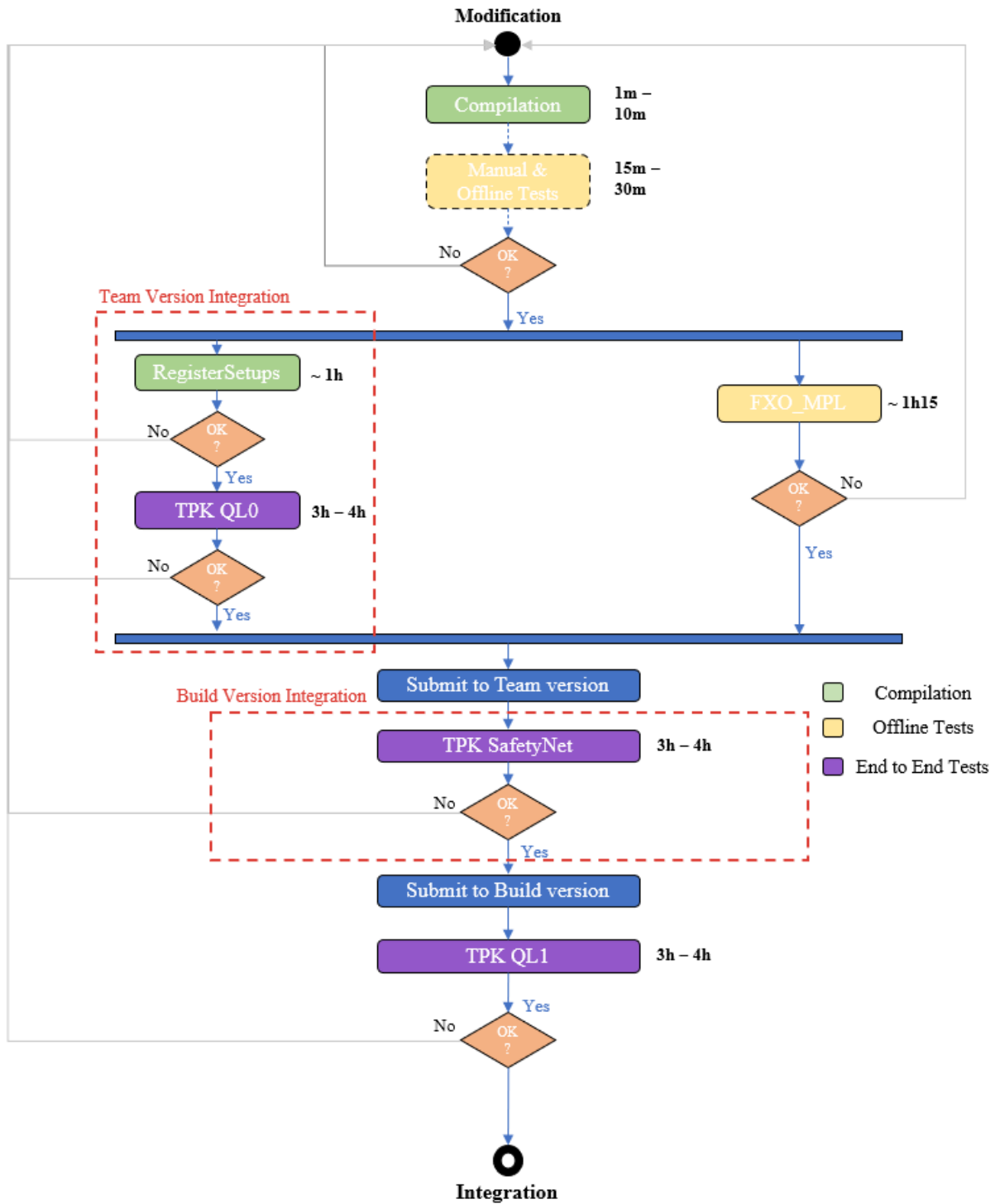


FIGURE 2.2 – Workflow

## 2.3 L'application MX.3

Avant toute chose, il est important de comprendre le fonctionnement de l'application MX.3. L'application MX.3 a une multitude de fonctionnalités séparés en modules. Pour l'équipe FXO, celle-ci va développer plusieurs modules. Un exemple de module est le pricing. Ce module permet de créer des options sur devise et générer un ordre sur cette option. Un deuxième module est la simulation. Ce module simule l'évolution du prix dans le temps d'un contrat qui a été généré précédemment. De manière générale, ces modules sont partagés sur des classes d'actifs différentes (actions, obligations, dérivés sur les obligations, ...).

L'application est construite à l'aide de 3 éléments principalement :

- Un ensemble de dynamic linked libraries (dll) qui sont des fichiers générés à travers la compilation du code source en C++ contenant ses classes et fonctions
- Une connexion à une base de donnée
- Un environnement MX

L'environnement est personnel à chaque utilisateur de l'application. Il permet de définir des paramètres de l'application pour une meilleure facilité d'utilisation. Par exemple notre équipe possède son propre environnement afin d'attacher un debugger à celle-ci et faciliter la découverte de bug dans le code. De manière similaire, chaque TPK possède son propre environnement également.

Du côté des clients l'application communique avec différents serveurs qui ont chacun un rôle bien spécifique. Tout d'abords, l'application est présente chez les clients à travers une interface utilisateur écrite en Java. Elle ne contient aucune logique qui elle est se situe dans la couche business des serveurs constitués de moteurs de calcul et une couche de séquençage des données (middleware). Enfin, cette couche communique avec une base de données contenant les données de marché.

## 2.4 Présentation de la problématique

### 2.4.1 Code Legacy

MX.3 est un logiciel qui a bénéficié de plus de trois décennies de développement technologique. L'essentiel de la logique de cette application (ou modèle) est programmé en C++, un langage informatique qui, malgré son importance, reste relativement récent. Ce langage était destiné à introduire la programmation orientée objet au langage C. En effet, la première normalisation du C++ ne date que de 1998, soit douze ans après la création de Murex et ce n'est qu'en 2011 que le langage se distingue véritablement du langage C, avec l'introduction de fonctionnalités spécifiques.



Un point crucial du langage C++ est sa conception basée sur la compatibilité rétroactive, ainsi que sa compatibilité avec le langage C. En pratique, cela signifie que chaque nouvelle version du langage doit pouvoir comprendre et interagir avec ses versions antérieures, ainsi qu'avec le code écrit en langage C.

Ce principe de compatibilité a un avantage majeur : il facilite considérablement la transition d'une version antérieure du C++ vers une version plus récente. Cependant, cela entraîne également un inconvénient significatif pour des logiciels comme Murex. Au fil du temps, le code produit devient de plus en plus obsolète, c'est ce qu'on appelle couramment « code legacy ». Cela peut poser des défis en matière de maintenance, de performance et de sécurité.

Dans le cadre de mon stage, un exemple de code legacy dans le module de pricing est l'utilisation d'un objet appelé « FinDefi ». Cet objet représente l'ensemble des informations liés à un change (l'euro-dollar noté EUR/USD par exemple). Il contient non seulement des informations statiques sur ce changes (devises, dates) que des données de marché dynamiques (le cours du change), ce qui en fait un objet volumineux et complexe. Petit à petit, d'autres objets sont venus le remplacer mais celui-ci reste extensivement utilisé notamment dans le code de pricing.

## 2.4.2 Importance des tests unitaires

L'équipe FXO est actuellement responsable d'une base de code C++, d'environ 700 000 lignes. Pour valider une modification du code, un ensemble de tests de l'application End to End sont exécutés. Ces tests, impliquent généralement le lancement d'une session MX et la connexion à une base de donnée distante. Ils représentent des cas d'utilisation de l'application. Par exemple, un utilisateur va se connecter à l'application, créer un ordre d'achat ou de vente sur une option sur devise et valider cet ordre.

Ces tests sont développés et maintenus depuis plusieurs années par les équipes de Quality Assurance (QA) et sont un élément essentiel pour pouvoir continuer à développer et maintenir une application de l'envergure de MX.3.

Cependant, étant donné que la priorité de Murex a toujours été de garantir que l'application livrée à ses clients soit aussi exempte de bugs que possible ces tests se sont multipliés et leur temps d'exécution est devenu inadapté dans le cadre d'un développement agile qui nécessite une itération rapide de son processus de développement. En effet, ces tests sont pour la plupart du temps le premier retour qu'un développeur reçoit suite à la modification de son code. Ils ont de plus l'inconvénient d'être lancés qu'à la suite d'une routine d'une heure évoqué précédemment permettant de créer les environnements de tests sur des machines distantes.

L'avantage de ces tests est qu'ils permettent de générer un grand nombre de cas

d'utilisations assez complexes. Cela permet de couvrir le code MX, non seulement dans les scénarios d'utilisation les plus courants mais également de couvrir les situations plus particulières (corner case).

Des tests plus simples, appelés tests unitaires, peuvent être mis en œuvre sans avoir à utiliser ce processus. L'accent est mis de plus en plus sur le développement et la maintenance de ces tests. Ces tests se concentrent sur le comportement d'une fonction lorsqu'elle est appelée avec un certain ensemble de paramètres et sont indépendants d'une session MX et de la base de données.

Ces tests sont bien plus rapides à exécuter et sont conçus pour échouer facilement dans les cas d'utilisation les plus simples afin de fournir un retour rapide au développeur de l'ordre de la minute contre l'heure pour les tests End to End. De plus, une différence majeure est qu'ils ne nécessitent pas de connexion à la base de données ni d'une session MX active, car ils ne testent que l'appel à une fonction. Ils sont de ce fait désignés comme tests offline par opposition aux tests End to End qui sont online.

Néanmoins la mise en œuvre de ces tests peut présenter certaines difficultés qui proviennent notamment du fait que l'on n'accède ni aux bases de données distantes ni à une session de l'application MX.3. De plus, comme ceux-ci sont bien moins nombreux, ils ne constituent pas une réelle barrière aux changements défectueux du code. Conceptuellement, ceci est représenté par une pyramide de tests qui est inversée.

### 2.4.3 Enoncé de la problématique

Le module de pricing de l'équipe FXO est un exemple où l'écriture de tests unitaires est plus complexe. Étant donné que celui-ci utilise extensivement un objet qui est la FinDef et que cet objet nécessite une connexion aux bases de données Murex pour être initialisé, cela rends l'écriture d'un test unitaire impossible lorsque cet objet est appelé. Néanmoins, Murex a développé plus récemment un framework appelé MxFTest<sup>†</sup> permettant de simuler des modules de l'application sans avoir à créer une instance complète de celle-ci. Il s'agira donc de développer des tests unitaires extensifs à l'aide de ce framework.

De ce fait, la problématique à laquelle nous allons nous intéresser pendant ce stage est alors la suivante :

**Développer les tests unitaires du module de pricing d'options de devises et améliorer son architecture à l'aide de ces tests**

## 2.5 Cahier des charges et objectif

Lors de ce stage, il s'agira donc de développer une bonne base de tests d'intégration pour le module de pricing. Les critères de développement de ces tests peuvent être résumés en trois

axes principalement.

- Couvrir un large panel d'options (vanilles, asiatiques, barrières, ...)
- Multiplier les réactions (changement de nominal, de strike<sup>†</sup>, de maturité, réaction à une réaction, ...)
- Couvrir les vérifications de résultats (vérifier systématiquement les premiums<sup>†</sup>, les sensibilités<sup>†</sup>, les données de marché, ...)

Une autre approche dans la manière de développer les tests est le Test Driven Development. Elle consiste à modifier le code et développer en boucle des tests de plus en plus exhaustifs permettant de couvrir cette modification bien spécifique du code. Avec les membres de l'équipe, nous avons décidé qu'il était plus intéressant d'axer le développement des tests sur les deux premiers critères dans un premier temps puis d'effectuer du Test Driven Development lorsque la base de tests sera non négligeable.

Également, il est possible de caractériser la manière à laquelle est écrite un test :

- Les tests statiques. Dans un test statique, l'ensemble des entrées, réactions et sorties du test sont statiques. Ces tests permettent de s'assurer que le code exécute de façon précise nos attentes. Ils permettent d'être exigeants quant à la qualité du code fourni.
- Les tests dynamiques. Ces tests relâchent les exigences des tests statiques sur les entrées et sorti afin de gagner en abstraction et donc en temps de développement. Ils sont particulièrement adaptés pour les réactions. Par exemple, la réduction du nominal d'un certain facteur doit systématiquement se traduire par la réduction du même facteur dans le prix de l'option pour la majorité des options.
- Les tests de stabilité de comportement. Ces tests permettent de relâcher encore les exigences d'un test dynamique. Dans ce type de tests, il s'agira uniquement de vérifier qu'il n'y a pas eu de modification par rapport aux valeurs de référence sans forcément se soucier de la justesse de ces derniers. Ces tests ont l'avantage de pouvoir être développés plus rapidement mais obstruent la compréhension générale du code de tests.

Il a également été décidé que l'attention du développement des tests se portera principalement sur des tests statiques combinés à des tests dynamiques afin d'assurer une meilleure maintenabilité des tests.

Une métrique importante à prendre en compte lors du développement de tests est la couverture du code. La couverture d'un test représente la partie du code que ce test exécute et plus spécifiquement les lignes que le test couvre. Plus la couverture du test est grande, et plus aisément les erreurs seront détectées dans le code. Toutefois cette métrique n'est pas parfaite car elle ne permet pas de représenter la multiplicité des scénario d'exécution d'une ligne de code. Néanmoins, cette métrique permettra en parti d'évaluer l'avancement de mon projet car cela reste une mesure assez représentative et usuelle de l'efficacité d'un test.

Actuellement, le pourcentage de couverture de tests unitaires du module de pricing est très faible (entre 5% et 10% pour certain fichiers). De plus, bien que certaines parties du

codes de pricing soient touchées par les tests unitaires, des tests spécifiquement écrits pour le pricing sont presque inexistants.

## Chapitre 3

# Développement de ma mission

### 3.1 Options sur devises

Il est usuel dans les marchés financiers de créer des contrats sur un actif sous-jacent pour des raisons multiples comme la spéculation ou la gestion du risque. Ces contrats sont appelés des dérivés car ils dépendent toujours d'un (ou plusieurs) actifs. Je vais détailler ici le fonctionnement de certains de ces contrats sur lesquels j'ai développé mes tests unitaires.

#### 3.1.1 Options Vanilles

Les options les plus simples sont les options vanilles. Il en existe deux types : les call et les put. Les call donnent à leur détenteur l'option d'acheter la devise sous-jacente à une certaine date future (ou maturité) et à un certain prix (appelé strike) déterminés à l'avance. Si le cours de cette devise (appelé  $\text{spot}^+$ ) à maturité est supérieur au strike, alors le détenteur exerce son option et achète l'actif au prix du strike. Il peut alors revendre immédiatement cet actif au prix du marché et garder la différence.

Plus formellement, en notant par  $S$  le spot,  $K$  le strike, le flux monétaire qui se produit lors de l'exercice de cette option est  $(S - K)_+$ . Pour une option européenne, l'exercice de l'option ne peut se produire qu'à sa maturité  $T$ . Pour une option américaine, l'exercice peut se produire à n'importe quel moment entre l'émission du contrat et sa maturité.

Ainsi, un call est une option qui, si achetée, permet de spéculer à la hausse d'un actif. Le détenteur de cette option prévoit que le prix de l'actif sera supérieur au strike lors de l'exercice de l'option.

Les put, sont le mécanisme inverse des call. Ils donnent à leur détenteur l'option de vendre la devise sous-jacente. On peut alors noter qu'un call dans une devise A payé en devise B est un put dans une devise B payé en devise A.

### 3.1.2 Options Barrières Simples et Doubles

Une option barrière est un paramètre supplémentaire qui se place sur une option vanille afin de construire un produit plus complexe. Les barrières ont pour but d'affiner la spéculation sur l'évolution du prix d'un actif. Cet affinage de la prédiction permet de réduire le prix de l'option barrière par rapport à une option vanille simple.

Une barrière est un niveau de prix du sous-jacent qui si atteint provoque l'activation (barrière IN) ou la désactivation (barrière OUT) du contrat. Le déclenchement d'une barrière se produit lorsque le prix du sous-jacent franchit à la hausse (UP) ou à la baisse (DOWN) cette barrière. Il est également possible de placer une deuxième barrière, dans quel cas, le sens de franchissement n'a pas d'importance est le contrat est activé (IN) ou désactivé (OUT) si le prix franchit une des deux barrières.

### 3.1.3 Options Touch

Les options Touch permettent d'émettre un flux monétaire lorsqu'un niveau de prix du sous-jacent (ou barrière) est atteint. Le sens de franchissement peut être à la hausse ou à la baisse comme pour les options barrières.

### 3.1.4 Options Knock-In Knock-Out

Les options Knock-In Knock-Out (KIKO) sont des options très similaires aux double barrières car elles possèdent également deux niveaux de prix. De la même manière que pour les barrières, ces options viennent paramétrer une option vanille plus simple à l'aide de deux niveaux de prix. Un premier niveau de prix est le niveau knock-in (KI) qui permet d'activer l'option. Le niveau knock-out (KO) quant à lui permet de la désactiver. De manière générale, la condition d'activation de l'option est d'atteindre le knock-in sans atteindre le knock-out.

### 3.1.5 Options Asiatiques

Les options asiatiques sont très similaires aux options vanilles à la différence que celles-ci se basent sur des moyennes du spot et non le spot lui même. Elles permettent de réduire la sensibilité à la volatilité des options vanilles.

## 3.2 Sensibilités des options

Le prix d'une option dépend de plusieurs facteurs qui évoluent au cours du temps. Pour quantifier ces dépendances, il est usuel de calculer ces sensibilités. Les sensibilités que j'ai pris

en compte dans mes tests unitaires sont les suivantes :

- **Delta** :  $\Delta = \frac{\partial P}{\partial S}$ , la sensibilité du prix de l'option par rapport au cours du sous-jacent.
- **Gamma** :  $\Gamma = \frac{\partial^2 P}{\partial S^2} = \frac{\partial \Delta}{\partial S}$ , la sensibilité du delta par rapport au cours du sous-jacent.
- **Vega** :  $\nu = \frac{\partial P}{\partial \sigma}$ , la sensibilité du prix de l'option par rapport à la volatilité du sous-jacent.
- **Vanna** :  $\frac{\partial^2 P}{\partial S \partial \sigma} = \frac{\partial \Delta}{\partial \sigma} = \frac{\partial \nu}{\partial S}$  la sensibilité du Delta par rapport à la volatilité, et la sensibilité du Vega par rapport au cours du sous-jacent.
- **Volga** :  $\nu = \frac{\partial^2 P}{\partial \sigma^2} = \frac{\partial \nu}{\partial \sigma}$ , la sensibilité du Vega par rapport à la volatilité du sous-jacent.
- **Theta** :  $\Theta = -\frac{\partial P}{\partial \tau}$ , la sensibilité du prix de l'option par rapport au temps écoulé depuis la création de l'option.
- **Rho** :  $\rho = \frac{\partial P}{\partial r}$ , la sensibilité du prix de l'option par rapport au taux d'intérêt sans risque d'une des devises du contrat.

### 3.3 Présentation de l'architecture actuelle du code de Pricing

Le module de pricing est l'un des plus anciens module de l'équipe FXO. Celui-ci a beaucoup évolué dans le temps, mais comme évoqué précédemment, il contient encore beaucoup de code legacy (code hérité). L'évolution du code pricing fait l'objet d'une road map dans l'équipe FXO qui est référencée dans le wiki interne. C'est un projet qui est central pour l'évolution de l'application MX.3.

De façon concrète, le pricing consiste à charger un ou plusieurs tableaux (appelés leg) contenant des informations sur l'option (appelé deal). Au sein de chaque leg, on trouve un orchestrateur. Cet objet, permet de gérer les réactions provenant directement de l'interaction utilisateur avec la leg. Par exemple, un utilisateur va décider de changer le strike d'une option vanille. Ce changement va alors provoquer des réactions de mise à jour de plusieurs valeurs (comme le prix de l'option) qui sont gérés à l'aide de l'orchestrateur. Un orchestrateur est donc comme son nom l'indique l'objet qui garantit la mise à jour de l'ensemble des éléments d'une leg.

Au sein de l'orchestrateur, on va également trouver un objet qui est l'évaluateur. Cette objet récupère de l'orchestrateur et garde en mémoire l'ensemble des informations nécessaires pour évaluer une option. Il contient des données statiques (gardés dans un objet nommé instrument) et des données dynamiques (gardés dans la mutableData<sup>†</sup>). L'utilisation de la FinDef dans le pricing se trouve au niveau de ces deux objets dans le pricing est c'est donc à ce niveau que les efforts sont concentrés afin de se débarrasser petit à petit de cet objet et de le remplacer par des objets plus récents comme l'instrument et la mutableData.

Un autre module à prendre en compte est le module de simulation. À l'instar du pricing, il permet d'évaluer et prédire le prix d'options sur une certaine échelle de temps. Toutefois

la grande différence entre ces deux modules est que le pricing est un module qui permet de construire et modifier le deal en question alors que la simulation se fait sur un deal fixé. La simulation n'a donc pas besoin de s'occuper de l'orchestration des éléments de son deal car celui-ci est déjà fixé. Ainsi, l'équipe fxo a pu remplacer l'utilisation de la FinDef par la mutableData dans ce module plus aisément que dans le pricing.

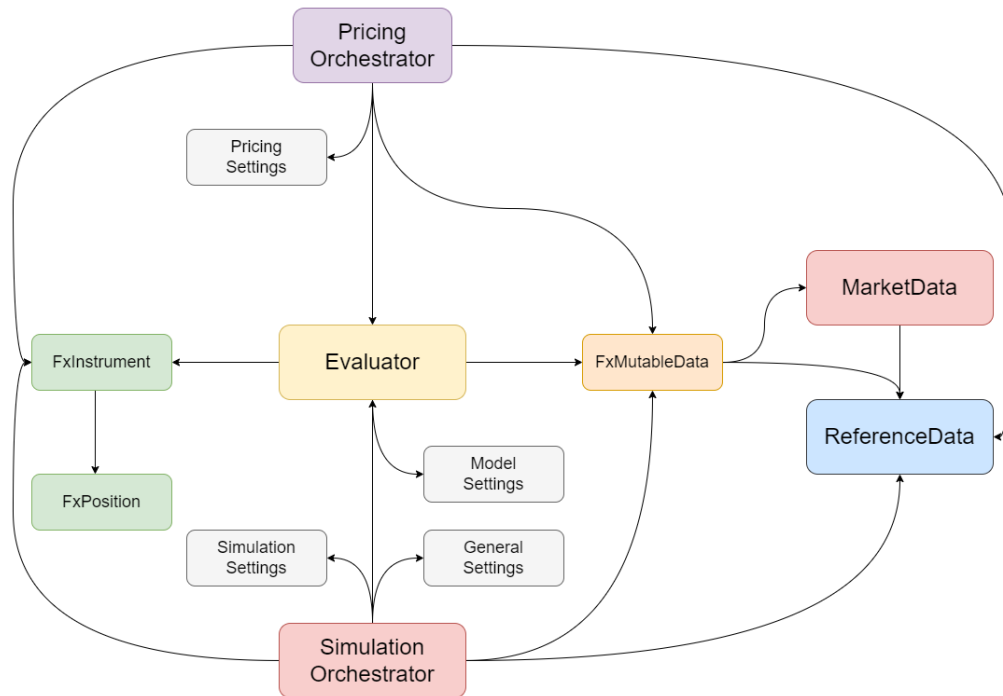


FIGURE 3.1 – Schéma des modules de pricing, simulation et évaluation

### 3.4 Organisation de mon travail et gestion de mon projet

Comme évoqué dans les chapitres précédents, Murex et l'équipe FXO en particulier s'organisent à l'aide du framework scrum et de la méthodologie agile. L'implémentation de cette méthode à Murex est très avancée et elle m'a permis d'organiser mon projet de stage de manière efficace. Tout d'abords, à mon arrivée, une epic a été créée afin que je puisse y insérer des stories. Cela allait permettre de ségementer et d'organiser mon travail qui reste un projet suffisamment indépendant du travail des autres développeurs.



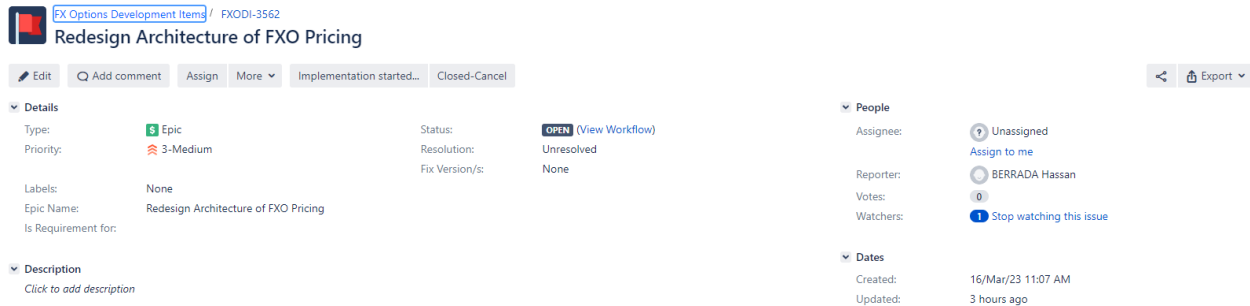


FIGURE 3.2 – Epic du Redesign de l’architecture de pricing d’options sur devises

Ainsi, la création de tâches plus élémentaires (ou stories) se faisait toujours en concertation avec mon tuteur et celles-ci étaient assignées à cette epic. La figure suivante montre une story. On peut y voir plusieurs information comme le type de la story, sa priorité, le lien à l’epic, les personnes assignées, etc... Il était également possible d’y insérer une estimation du temps que cela me prendrait et que l’on appelle dans le framework scrum story points<sup>†</sup> et qui représentait le nombre de jours que la story devrait me prendre.

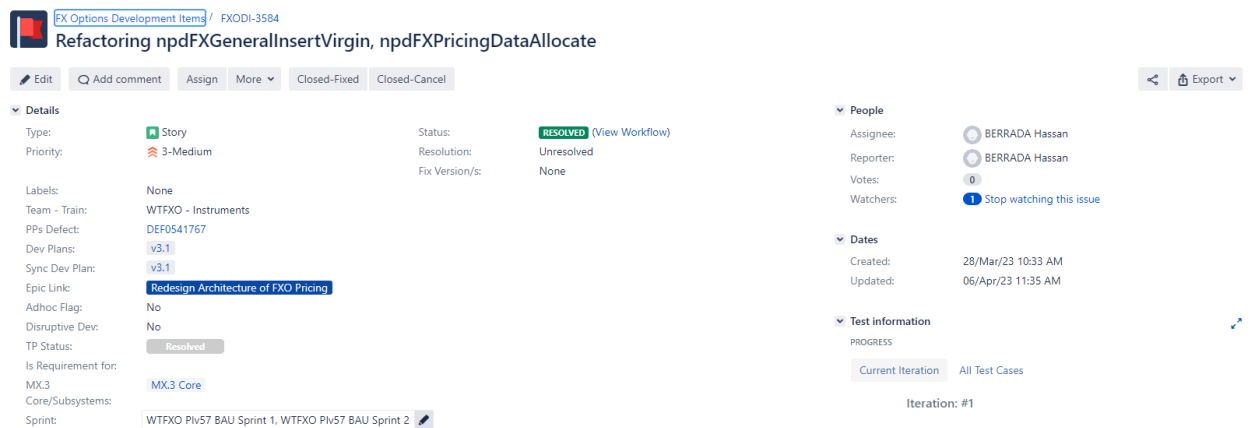


FIGURE 3.3 – Exemple de story

## 3.5 Premières missions de refactoring de code

Murex possède un ensemble d’outil de développement qui est très large. Afin de prendre en main ces outils, les premières missions que j’ai effectué lors de mon stage étaient des missions de refactoring de code assez simples. Les premières constituait uniquement à renommer/déplacer certaines parties du code puis ensuite de modifier certaines fonctions. Cela m’a permis de me familiariser à l’utilisation efficace des éditeurs de texte sur Visual Studio, à la gestion de mes modifications sur Perforce, à l’intégration du code, mais surtout à l’utilisation d’un debugger.

Un debugger est un outil permettant de s'attacher à un processus et permettant de suivre l'exécution du code ligne par ligne au fur et à mesure que le processus s'exécute. Le processus en question est une instance de l'application Murex. Ainsi, il est possible d'interagir avec l'application et de voir dans le code la pile d'appel des fonctions utilisées. Auparavant, je n'avais que très peu utilisé ce type d'outil, mais cet outil s'avère indispensable lorsque l'on traite une base de code aussi large que celle de Murex.

Les premières missions de refactoring étaient simples. Elles ont consisté à nettoyer certaines parties du code relative au pricing. Plus précisément, cela se rapportait à la création d'une leg. Ce nettoyage consistait principalement à renommer des variables, séparer et déplacer le code utilisé par l'équipe de fx cash de celui de notre équipe fx options, mais également à aligner certaines méthodes de l'orchestrateur.

Une fonctionnalité centrale du pricing est la fonction de *load* de l'orchestrateur. Celle-ci permet de créer 4 évaluateurs et de les remplir avec des données de marché (mutableData) différentes. Il a d'abord été question de renommer cette méthode dans les parties du code où celle-ci est évoquée. Cette méthode possède une méthode nommée buildEvaluators. Elle permet de classifier et construire les évaluateurs en fonction du type d'orchestrateur (vanille, barrière, asian, ...). Un orchestrateur Vanille permettra la construction de 4 évaluateurs Vanille.

La méthode buildEvaluators était propre à chaque orchestrateur. Sauf que pour la plupart des orchestrateurs (vanille, barrière, compound, ...), le code était très similaire ce qui créait de la duplication dans le code. La première tâche que j'ai effectué était d'aligner la méthode buildEvaluators sur l'ensemble des orchestrateurs. En programmation orientée objet, cette opération est une extraction dans la classe mère et permet de rendre le code plus facilement maintenable en éliminant la duplication et uniformisant le code.

Une autre tâche que j'ai effectué consistait à séparer l'utilisation de la fonction npdFXGeneralInsertVirgin, npdFXPricingDataAllocate et npdFXPricingDataInitialize. Ces fonctions étaient partagées avec l'équipe fx cash car le code des équipes fx options et fx cash était confondu auparavant. Elles permettent lors de l'ouverture du eTradePad<sup>†</sup>, d'initialiser une leg en créant les objets nécessaires pour cette leg comme l'orchestrateur. Toutefois avec l'évolution de l'application, ces deux équipes partagent de moins en moins de code et les dépendances entre les deux codes peuvent provoquer des difficultés supplémentaires dans le développement. C'est pourquoi la séparation du code de ces deux équipes est importante.

A l'issue de ces missions (voir Annexes A.1), je me suis habitué à utiliser les outils de développement de Murex dans la gestion des modifications, des versions et de l'intégration ainsi que l'utilisation du debugger. Je commençais également à me familiariser à la stack de création d'une leg et de l'orchestrateur qu'elle contient.

1. Vanilla Option FXD	
Currency pair	EUR/USD
Buy/Sell	<b>Sell</b>
Call/Put	<b>Call EUR / Put USD</b>
Expiry	<b>Fri 26/05/17 (1y)</b>
Cutoff	NYC - 10:00 New York Time
Strike	<b>1.0993</b>
Traded nominal	1,000,000 EUR
⊕ Settlement details	
Hedge type	No Hedge
Spot	1.0500 / <b>900</b>
⊕ Swap points	-31.60
ATM Volatility	8.27 / <b>47</b>
25% RR   Fly	-1.719 0.354
Price quotation	USD pips
⊕ Spt/Fwd prem.	Spot
Trader Volatility	7.98 / <b>8.19</b>
Trader Price (USD pips)	298%
Trader Price (USD)	<b>R</b> 29,823
stg.fx.model	Black Scholes
stg.fx.Smile	On
Margin (USD pips)	0
Margin (USD)	0
Client Volatility	8.19
Client Price (USD pips)	298%
Client Price (USD)	<b>R</b> <b>29,823</b>
Spot Delta (% EUR)	46
Spot Delta (EUR)	-459,070
Fwd Delta (% EUR)	46
Fwd Delta (EUR)	-460,456
Vega (USD)	-4,320
⊖ Trader Greeks	
Trader Spot Delta (EUR)	-407,516
Trader Fwd Delta (EUR)	-408,723
Trader Gamma (EUR)	-54,601
Trader Vega (USD)	-4,190
Trader Vanna (EUR)	-17,387
Trader Volga (USD)	-45
Trader Theta (USD)	52
Trader Rho (Rho USD in...)	-0
Trader RhoF (Rho EUR i...)	4,361

FIGURE 3.4 – Visuel d'un Tradepad sur l'application Murex

## 3.6 Écriture des tests unitaires offline

La première partie de mon stage, m'a permis non seulement de me familiariser avec l'environnement de développement mais également de comprendre la nécessité d'une bonne base de tests offline contenant les tests unitaires. En effet, la validation de mes changements à travers la pipeline d'intégration pouvait prendre un temps considérable justement à cause de l'absence de tests unitaires offline pour le pricing. La première barrière fonctionnelle à une erreur introduite dans le code était alors les TPKs qui prenaient entre 1h et 4h pour les premiers retours. L'écriture de tests offline allaient alors permettre de réduire considérablement ce premier retour à quelques minutes voire secondes.

La première étape des tâches que j'ai effectué dans cette partie (voir Annexes A.2) était de mettre en place les premiers tests pour une leg d'option vanille. Pour ce faire, j'ai essayé de répliquer étape par étape dans mon test, l'initialisation d'une leg. Cette initialisation se fait en 4 parties essentielles où il faut renseigner des champs dans chaque partie. Premièrement, il faut renseigner le contrat (EUR/USD par exemple), la position (Buy/Sell), le payout (Call/Put), et enfin la maturité (1y). D'autres champs nécessaires sont fixés automatiquement comme le nominal (à 1 000 000 €) et le strike (au spot actualisé).

Le challenge de cette partie est l'utilisation du Framework MxFTest sur le pricing. En effet, des tests de pricing propres à l'équipe fxo n'existaient pas hormis un test repris de l'équipe fx-cash par Elom. Il fallait donc s'approprier cet environnement pour l'écriture de tests qui conviennent à notre équipe. Ainsi, ma première tâche était de comparer cet environnement à l'environnement d'une application MX et de comprendre les similarités et différences. L'idée était donc de trouver les éléments nécessaires à fixer dans mon test afin que celui-ci se rapproche le plus possible d'une leg de pricing. Cela permettrait de facilement en écrire car, l'interface graphique de l'application permettra de voir directement les valeurs de références et de s'en rapprocher dans notre environnement de test (Voir Annexe A.4 pour un exemple).

Toutefois, les différences entre ces deux environnements font que les valeurs ne vont pas être précisément les mêmes, mais je m'attendais à avoir des valeurs assez consistantes et dans les mêmes ordres de grandeur pour une même option. La première différence entre les deux environnements représente les données statiques. Dans l'application, elles sont lues de la base de données et sont écrites dans un objet lorsque l'application est allumée. Dans l'environnement de tests, elles sont lues d'un fichier json contenant ces informations. On y retrouve, les devises, les paires de devise, ainsi que des informations sur les courbes de taux à utiliser. La deuxième différence vient des données dynamiques qui sont de la même façon que pour les données statiques lus d'une base de données dans l'application et dans un fichier json dans l'environnement de test. Elles contiennent les courbes de taux, les cours, les forward<sup>†</sup>, les courbes de volatilité et d'autres informations sur une paire de devise.

Enfin, la dernière différence concerne les configurations utilisateurs. Dans l'environnement

de l'application, elles sont construites à l'aide de la base de données et sont propres à chaque utilisateur de l'application. Toutefois dans l'environnement de tests, elles ne sont pas fixés. Ceci empêche de nombreuses fonctionnalités dans les tests comme l'utilisation du mode Bid/Ask. Par défaut le mode est en Mid qui a une fonction informative. Le mode Bid/Ask est plus utile car le prix Bid représente le prix d'achat et le prix Ask, le prix de vente. D'autres configurations comme l'utilisation de la smile de volatilité (ou smile<sup>†</sup>) afin d'obtenir une meilleur estimation du prix d'une option se trouvent dans la configuration utilisateur.

C'est en développant mes tests que je me suis rendu compte de ces différences. Ainsi, les premiers tests que j'ai développé était des tests très simples de pricing sur certaines options. Ces options sont les options vanilles, les options barrières, les options touch, les options KIKO et les options asiatiques. J'ai donc créer une classe de test que j'ai nommé *NpdIntegrationTest*. Cette classe contient les principales fonctions permettant la mise en place des tests comme l'initialisation des données statiques, des données dynamiques et des paramètres de l'option. Je l'ai ensuite étendue sur l'ensemble des autres types d'options (Barrières, Asian, ...) afin de créer pour chaque classe une initialisation bien spécifique. Enfin, toutes ces classes servent à créer des *fixtures*<sup>‡</sup> qui permettent d'aligner l'ensemble des tests sur une certaine configuration de *fixtures* qui peut être réutilisée.

### 3.7 Mise en place de l'extension TestMate

Lorsque j'ai commencé à développer mes tests offline unitaires, l'équipe m'a présenté un outil pour pouvoir interagir avec les tests unitaires qui s'appelle TestMate. Cet outil est une extension sur visual studio permettant d'exécuter et debugger des tests ainsi que de voir leur état.

Le principale désavantage de cet outil est le fait qu'il n'est pas directement utilisable et nécessite des configurations à chaque fois que l'on change de version. De plus les tests doivent être construits à chaque version. Afin de faciliter l'utilisation de cette extension, j'ai écrit un script en python permettant d'automatiser ce travail. Cela allait pouvoir m'aider à l'avenir car je savais que j'allais développer extensivement les tests lors de mon stage.

Ce script permet de charger sur l'environnement de test TestMate et de construire les tests que l'on spécifie en ligne de commande. Par exemple une ligne de commande typique peut-être :

```
1 $ python script_for_testmate.py -k "pricing|simulation" -d "orchestration"
```

Cette ligne de commande permet de simplement charger et construire les tests de pricing et de simulation sans les tests contenant l'orchestration.

L'interface que l'on obtient de TestMate est très simple d'utilisation, on gagne l'avantage de facilement voir l'état de nos tests mais surtout de les debugger. Cette extension m'a permis

par la suite de facilement interagir avec mes tests (Voir Annexe A.5) . Il m'était possible de voir l'état de chacun de mes tests lorsque j'apportais une modification au code. De même je pouvais facilement debugger le code de test car les champs que je testais étaient les champs les plus importants pour le pricing comme le premium, les données de marchés (le spot, la volatilité, ...).

## 3.8 Deuxièmes missions de refactoring de code

Suite à la mise en place de mes tests, je me suis alors de nouveau attaqué à du refactoring de code. Pour cette deuxième partie, nous avons décidé avec mon tuteur que j'allais me concentrer plutôt sur les options vanilles quanto. En effet, ces options, sont parmi les dernières options du code pricing utilisant la FinDef dans la configuration. Une option quanto est une option dont le payout  $(S_T - K)_+$  est payé dans une devise tierce. Cette option permet de s'exposer à des options sur devise étrangères, en se couvrant du risque de taux de change entre la devise tierce et la devise du payout. Par exemple, si l'on est une banque anglaise, il est possible d'acheter un call (EUR/USD) en fixant par avance le taux de change du payout (en EUR/GBP ou GBP/USD).

La fonction que l'on a ciblé spécifiquement est la fonction `fxmSmpGetQuantoAttributes`. Cette fonction est appelée dans le load de l'orchestrateur et permet de récupérer les attributs de la quanto (comme les volatilités de la quanto par rapport aux autres devises, la courbe de taux de la quanto, ...). Cette fonction possède deux cas distinct. Dans le contexte de la simulation, elle va récupérer ces attributs à partir de la mutableData et dans le contexte du pricing, elle les récupérera de la FinDef. Voici un pseudo code de cette fonction :

```
1 void fxmSmpGetQuantoAttributes(...)
2 {
3     if(simulationContext && quantoData) // if simulationContext and
↪   quantoData exists
4     {
5         // Get the quanto attributes from the mutable Data
6     }
7     else // if pricing context or quantoData does not exist
8     {
9         // Get the quanto attributes from the findef
10    }
11 }
```

Le but était de pouvoir ne plus avoir à utiliser la findef dans le contexte du pricing. Idéalement, la fonction n'aurait plus de différenciation entre le contexte de la simulation et du pricing. A l'issue de mes modifications, il a été possible d'unifier les codes de la manière suivante :

```
1 void fxmSmpGetQuantoAttributes(...)
2 {
```

```
3     if(quantoData) // if quantoData exists
4     {
5         // Get the quanto attributes from the mutable Data
6     }
7     else // if quantoData does not exist
8     {
9         // Get the quanto attributes from the findef
10    }
11 }
```

Dans ce deuxième cas, le code de pricing et de simulation est unifié mais dans les cas problématiques où la quantoData n'existe pas, on passe toujours dans le code où l'on récupère les informations de la findef. Ces cas se présente notamment lorsque la devise de la quanto est l'une des devises du contrat (on appelle ce type de contrat des self quanto et sont très peu utilisées).

En parallèle de ces modifications de code, j'ai alimenté mes tests unitaires avec des tests de pricing d'option quanto. Ces tests m'ont permis de pouvoir modifier plus rapidement mon code car ils me permettaient de voir facilement si un changement avait un impact sur le pricing d'une quanto.

# Chapitre 4

## Retour sur mon expérience

### 4.1 Retour sur mes objectifs

A l'issu de ce stage, une bonne base de tests de pricing offline a été mise en place. La couverture du code de pricing a augmenté significativement. Sur certains fichiers celle-ci a doublé et représente quelques milliers de lignes de code. Cette couverture permettra à l'avenir de facilement et rapidement modifier le code de pricing (voir Annexes A.6 et A.7). En effet, les tests que j'ai écrits s'exécutent en quelques minutes pour donner un premier retour fonctionnel du pricing. J'avais commencé à en faire l'expérience lors de ma troisième partie de stage grâce à ma mission sur les options quanto. Toutefois, la couverture reste encore faible par rapport aux tests online et beaucoup d'erreurs ne seront pas saisies par ces tests. Ils ne peuvent donc pas remplacer pour l'instant les tests de pricing online.

Pendant la troisième partie de mon stage, j'ai voulu investir du temps sur l'écriture de tests unitaires offline de pricing ayant la possibilité d'avoir les modes de Bid/Ask et Smile On, évoqué précédemment. Mon argument étant que ces paramètres étaient cruciaux dans la plupart des legs de pricing. Ces modes là nécessitaient un objet de configuration qui était absent de mes tests. J'ai investi beaucoup de temps sur des tentatives de réplication ou de contournement de l'objet. Il était par exemple envisagé de surcharger au sein de l'environnement de test certaines fonctions de l'orchestrateur bien choisies afin de pouvoir l'alimenter avec les bonnes informations de configuration. Cela n'a pas aboutit mais a permis au moins de mettre en évidence une des faiblesses des tests offline.

### 4.2 Mission de développeur

Ce stage a été ma première exposition à un rôle de développeur et à une large base de code. De plus, cela était ma première expérience de code en C++. J'ai donc beaucoup appris sur le développement et le langage C++ pendant ce stage. J'ai par ailleurs eu l'occasion de bien comprendre la méthode agile et d'en voir l'application concrète au sein d'une entreprise de développement informatique.



## 4.3 Écueils que j'ai rencontré

Les différents écueils que j'ai rencontré lors de ce stage ont été riches en enseignements.

Premièrement, le fait que l'utilisation efficace d'un debugger était clé afin de pouvoir modifier un code de l'envergure de celui de Murex. Lors de mes précédentes expériences, j'étais principalement exposé à des codes en python assez court et indépendant. Toutefois, le code de Murex est bien plus grand et son exécution est complexe. Il faut arriver à cibler les parties du code à étudier ainsi que de bien isoler son problème. Ce sont des compétences que j'ai petit à petit développé lors du stage mais auxquelles je n'avais pas été exposé auparavant.

Deuxièmement, j'ai appris l'importance de cibler ces modifications quitte à faire des modifications atomiques qui seront validées petit à petit jusqu'à arriver à l'objectif attendu. En effet, au début de mon stage, il m'arrivait d'apporter des modifications trop importantes en une seule fois. Cela faisait que les problèmes que je résolvais par la suite n'était pas suffisamment ciblés. Par la suite, je commençais à modifier mon code de façon à m'attendre aux effets que cela pouvait engendrer. Cela m'aidait à mieux cibler les sources de problèmes et de les modifier.

Enfin, je trouve avoir pris beaucoup de temps à m'habituer à l'environnement de développement de Murex qui est très complexe. Bien comprendre le processus d'intégration du code, et apprendre l'utilisation des outils de Murex avait constitué un premier défi en soit. En rétrospective, j'aurai dû accentuer la première partie de mon stage à me former à ces outils. C'est principalement le cas pour l'utilisation de debugger. Après avoir fini l'écriture de mes tests et que ceux-ci soient intégrés en production, une pipeline de vérification de

# Conclusion

Cette expérience en développement fut très enrichissante chez Murex. J'ai eu l'occasion de rencontrer une équipe dynamique et très à l'écoute. De plus, notre équipe était très diverse culturellement et professionnellement et cela a été un plaisir de pouvoir travailler et échanger avec eux. Cet environnement de travail était idéale pour mon stage où j'ai pu y développer de solides compétences techniques en développement.

J'ai eu la chance lors de mon parcours aux Mines de m'être exposé à plusieurs domaines académiquement et professionnellement. Cela m'a permis de me forger une bonne base de compétence en mathématiques, informatique et science des données ainsi que de me donner la maturité nécessaire pour connaître les domaines dans lesquels j'aimerais pouvoir diriger ma carrière.

J'ai également pu tirer des enseignements très variés entre mes stages de science de données en césure et mon stage de développement informatique à Murex. En césure, j'ai beaucoup apprécié le côté analytique de mes stages tandis que dans mon stage de fin d'études, j'ai grandement appris de la gestion rigoureuse du développement d'une application à grande échelle et de l'importance de produire un produit de qualité à la visée d'un client.

Étant intéressé par la finance j'ai donc décidé de continuer mes études afin d'approfondir mes connaissances de ce domaine. Les compétences et enseignements que j'ai acquis pendant mon parcours me sont aujourd'hui très précieuse dans la poursuite de mes études.

# Glossaire

**buy side** Le sell side fait référence aux entreprises qui achètent des titres sur les marchés financiers et comprend les gestionnaires de placements (asset manager), les fonds de pension et les fonds spéculatifs (hedge funds). 7

**epic** Une grande unité de travail qui généralement est accomplie en plusieurs sprints. Elle regroupe plusieurs stories et permet de regrouper des travaux autour d'un thème ou d'une fonctionnalité. 8, 12, 23, 24

**eTradePad** Interface graphique de MX.3 permettant de créer un deal (un trade) sur de nombreux actifs (devises, actions, obligations, ...). 25

**FinDef** Cet objet représente l'ensemble des informations liés à un change (l'euro-dollar noté EUR/USD par exemple). Il contient des informations statiques sur ce change (devises, dates) et des données de marché dynamiques (le cours du change). Il nécessite une connexion à une base de données afin d'être initialisé. 16, 17, 22, 23, 29, 34

**fixture** Une fixture est la mise en place d'un environnement bien spécifique à chaque test qui assure que tout les paramètres d'une même fixture sont les mêmes. Cela permet d'avoir les mêmes conditions initiales pour chaque test et de fixer cet environnement. 28

**forward** Spot capitalisé avec la différence des taux d'intérêt directeur entre les deux devises soit  $F_T = S_t \exp(-(r_1 - r_2)(T - t))$ . 27

**mutableData** Objet permettant de stocker les données de marché d'un deal. C'est un des objets destiné à remplacer la FinDef. 22, 23, 25, 29

**MxFTest** Framework interne de Murex permettant de créer un environnement Mx de test offline qui se rapproche de celui de l'application online. 17, 27

**premium** Correspond au prix d'achat ou de vente d'une option. Il en existe de plusieurs types, comme le premium du client, du trader, ou du modèle. 18, 29

**sell side** Le buy side fait référence aux entreprises qui émettent, vendent ou négocient des titres, et comprend les banques d'investissement, les sociétés de conseil et les sociétés. 7

**sensibilité** Correspond aux dérivés partielles (simples ou multiples) du prix du l'option en fonction de facteurs qui évoluent au cours du temps qui influent sur ce dernier comme le cours du sous-jacent, la volatilité, le passage du temps et les taux d'intérêts neutres. 18

**smile** La smile de volatilité ou courbe de volatilité représente la volatilité implicite d'une option donnée par le modèle de Black-Scholes sur une option. On la représente souvent en fonction du strike par exemple. 28

**spot** Désigne le cours d'un actif. 20, 21, 27, 29

**story** C'est une fonctionnalité ou une tâche précise, dont l'utilité et la charge de travail est évaluée au préalable. 8, 12, 23, 24

**story point** Unité de mesure permettant de représenter la charge de travail d'une story (souvent associée à un jour-homme). 24

**strike** Désigne le prix d'exercice d'une option. Dans le cas des options sur devises, celui-ci correspond à un cours à atteindre pour que l'option soit active. 18, 20, 22, 27

**TPK** Ce sont les tests développés par Murex pour tester l'application MX.3 à travers son interface et sont considérés comme Tests End to End. 13, 15, 27

# Annexe A

## Annexes

### A.1 Tâches attribuées




















Assigned to Me				
T	Key	Summary	P ↓	
	FXODI-3517	Change the void* type of PricingOrchestrator's evaluators to Evaluator*		
	FXODI-3560	Changing the namespace from murex::fx::fxprc to murex::fx::opt::pricing		
	FXODI-3572	Rename the prPR_* structures in PricingOrchestrator.fwd.hpp		
	FXODI-3576	Remove of inputPricingOrchestrator, FxProduct and replace loadPricingOrchestrator by load		
	FXODI-3578	Make common Classify method in PricingOrchestrator		
	FXODI-3579	Make common fillImmutableData in PricingOrchestrator		
	FXODI-3584	Refactoring npdFXGeneralInsertVirgin, npdFXPricingDataAllocate		
	FXODI-3587	Make fillPricingContext final method of PricingOrchestrator		
	FXODI-3605	Moving int npdFXGeneralStdFunc from fx_nngen to fx_nopt		
	FXODI-3606	Deleting npdFXPricingDataInitializeOption		
1-10 of 26			1 2 3 ▶	

FIGURE A.1 – Tâches de la première partie

Assigned to Me			
T	Key	Summary	P ↓
	FXODI-3613	Moving npdFXGeneralInsertVirginOption and npdFXGeneralInsertVirginCash	
	FXODI-3618	Writing loading and pricing of vanilla option in mxftest	
	FXODI-3619	Make common buildEvaluator for basket and bestof	
	FXODI-3646	Writing loading and pricing of barrier option mxftest	
	FXODI-3647	Writing loading and pricing of asian option mxftest	
	FXODI-3648	Writing loading and pricing of vanilla strip option mxftest	
	FXODI-3649	Integrating the TestMate loader	
	FXODI-3659	Writing loading and pricing of rebate option mxftest	
	FXODI-3660	Writing loading and pricing of kiko option mxftest	
	FXODI-3661	Cleaning TestMate script	
11–20 of 26			◀ 1 2 3 ▶

FIGURE A.2 – Tâches de la deuxième partie














Assigned to Me			
T	Key	Summary	P ↓
	FXODI-3665	Set the nominal in NpdIntegrationTests	
	FXODI-3717	Finding ASAN issue in NpdIntegrationTests	
	FXODI-3730	Remove findef duplication in computeCore and fx_nsmpt	
	FXODI-3735	Writing loading and pricing of quanto option mxftest	
	FXODI-3738	Set BidAsk Mode in NpdIntegrationTest	
	FXODI-3744	Remove use of pricing context in fxmSmpGetQuantoAttributes	
21–26 of 26			◀ 1 2 3 ▶
Issues in progress			
T	Key	Summary	P ↓
	FXODI-3744	Remove use of pricing context in fxmSmpGetQuantoAttributes	
1–1 of 1			

FIGURE A.3 – Tâches de la troisième partie

## A.2 Comparaison de MX et de MxFTest

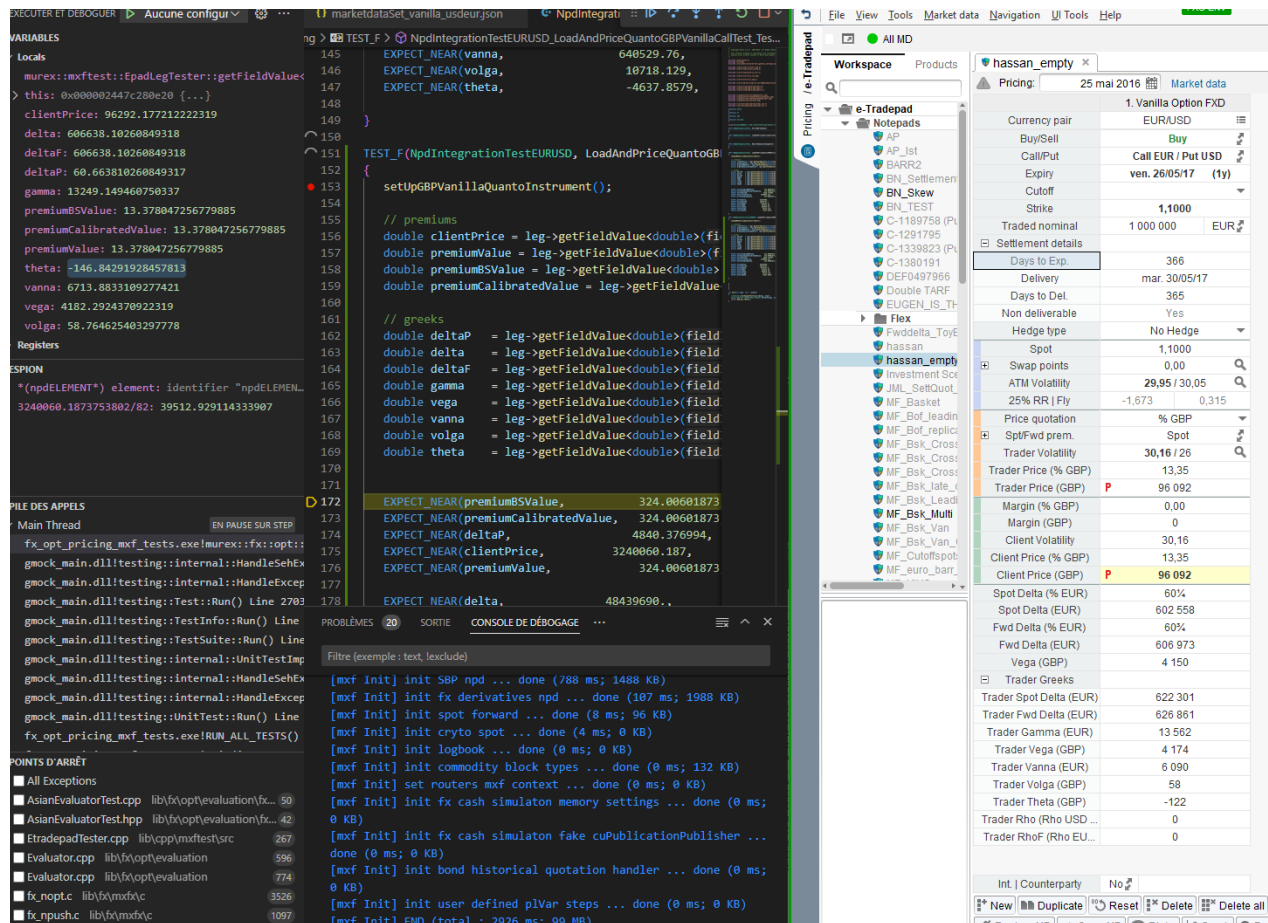


FIGURE A.4 – Instance de test à gauche contre instance de l’application à droite

## A.3 Script d'utilisation de Testmate

```

1  # For execution of the script, you can use:
2
3  # python3 script_for_testmate.py -k "pricing/evaluation/risk" -d "mxf"
4  # will look for fxo_test containing pricing or evaluation or risk and
   ↪ discard all fxo_test that contain mxf
5
6
7  import json
8  import os
9  import argparse
10 import re
11 import subprocess
12 import logging
13 import functools
14
15 class Logger:
16     """
17     This class is the Logger class. It sets the style of our logs.
18     """
19
20     def __init__(self, name):
21         self.logger = logging.getLogger(name)
22         self.logger.setLevel(logging.INFO)
23
24         self.logger.propagate = False
25
26         ch = logging.StreamHandler()
27         ch.setLevel(logging.INFO)
28
29         formatter = logging.Formatter('%(asctime)s] %(name)s] %(levelname)s]
   ↪ - %(message)s')
30
31         ch.setFormatter(formatter)
32
33         self.logger.addHandler(ch)
34
35     def info(self, message):
36         self.logger.info(message)
37
38     def warning(self, message):
39         self.logger.warning(message)
40

```



```

41     def error(self, message):
42         self.logger.error(message)
43
44
45 class LogDecorator:
46     """
47     This class helps with decorating our JsonWriter to log information on the
48     ↪ state of the execution of the script. It contains decorators for some
49     ↪ methods of the JsonWriter to help with logging.
50     """*
51
52     def __init__(self, logger):
53         self.logger = logger
54
55     def init_decorator(self, func):
56
57         @functools.wraps(func)
58         def wrapper(fxoTestJsonWriter, *args, **kwargs):
59             self.logger.info(f'Starting Loading Tests in TestMate')
60             func(fxoTestJsonWriter, *args, **kwargs)
61             return wrapper
62
63     def create_vscode_settings_file_decorator(self, func):
64
65         @functools.wraps(func)
66         def wrapper(fxoTestJsonWriter, *args, **kwargs):
67             self.logger.warning(f'VS Code settings file not found')
68             self.logger.info(f'Creating it')
69             func(fxoTestJsonWriter, *args, **kwargs)
70             return wrapper
71
72     def create_test_env_file_decorator(self, func):
73
74         @functools.wraps(func)
75         def wrapper(fxoTestJsonWriter, *args, **kwargs):
76             self.logger.warning(f'env.json file not found')
77             self.logger.info(f'Creating it')
78             func(fxoTestJsonWriter, *args, **kwargs)
79             return wrapper
80
81     def read_test_paths_from_fxotests_file_decorator(self, func):
82
83         @functools.wraps(func)

```

```

82     def wrapper(fxoTestJsonWriter, *args, **kwargs):
83         self.logger.info(f'Reading the existing tests from the htest file')
84         func(fxoTestJsonWriter, *args, **kwargs)
85         for test in fxoTestJsonWriter.test_paths:
86             self.logger.info(f"Test path found : {test}")
87     return wrapper
88
89 def build_target_decorator(self, func):
90
91     @functools.wraps(func)
92     def wrapper(fxoTestJsonWriter, *args, **kwargs):
93         target = args[0]
94         self.logger.info(f'The bat or exe file of the target: {target} was
95             ↪ not found')
96         self.logger.info(f'Building it ...')
97         func(fxoTestJsonWriter, *args, **kwargs)
98     return wrapper
99
100 def read_args_from_bat_decorator(self, func):
101
102     @functools.wraps(func)
103     def wrapper(fxoTestJsonWriter, *args, **kwargs):
104         self.logger.info(f'Reading args from bat files')
105         func(fxoTestJsonWriter, *args, **kwargs)
106     return wrapper
107
108 def
109 ↪ write_testmate_advanced_executables_in_vscode_settings_decorator(self,
110 ↪ func):
111
112     @functools.wraps(func)
113     def wrapper(fxoTestJsonWriter, *args, **kwargs):
114         self.logger.info(f'Writing the VS code settings file')
115         func(fxoTestJsonWriter, *args, **kwargs)
116     return wrapper
117
118 logger_decorator = LogDecorator(Logger('Loading Tests in Testmate'))
119 class FxoTestJsonWriter:
120
121     @logger_decorator.init_decorator
122     def __init__(self, args,
123         ↪ all_htests_file="/lib/fx/opt/test/common/htests.lf",
124         ↪ vscode_settings_file="/.vscode/settings.json"):

```

```

121
122     self.exec_args = args
123
124     self.version_path = os.getcwd()
125
126     self.all_htests_file = self.version_path + all_htests_file
127
128     self.vscode_settings_file = self.version_path + vscode_settings_file
129     if not os.path.isfile(self.vscode_settings_file):
130         self.create_vscode_settings_file()
131
132     self.env_name = ".mxvsce/env.json"
133     self.env_file = os.path.join( self.version_path, self.env_name )
134     self.settings_name = "settings.cmd"
135     self.build_settings_file = os.path.join(self.version_path, "build",
136     ↪ self.settings_name)
137
138     if not os.path.isfile(self.env_file):
139         self.create_test_env_file()
140
141     self.test_paths = []
142     self.list_target_paths = []
143     self.kept_target = []
144     self.exe_paths = []
145     self.bat_paths = []
146     self.env_args = []
147     self.dict_exe_args = {}
148     self.dict_path_num_tests = {}
149
150     return
151
152 def get_number_of_tests(self):
153
154     return len(self.exe_paths)
155
156 @logger_decorator.create_vscode_settings_file_decorator
157 def create_vscode_settings_file(self):
158
159     vscode_path = os.path.dirname(self.vscode_settings_file)
160     if not os.path.isdir(vscode_path):
161         os.makedirs(vscode_path)
162

```

```

163         with open(self.vscode_settings_file, 'w') as file:
164             json.dump({}, file)
165
166         return
167
168     @logger_decorator.create_test_env_file_decorator
169     def create_test_env_file(self):
170         """ Creating the env.json """
171
172         env_variables_output_file = os.path.join(self.version_path,
173             ↪ "build/env_output.txt")
174
175         subprocess.run(f'{self.build_settings_file} && set >
176             ↪ {env_variables_output_file}', shell=True,
177             ↪ stdout=subprocess.DEVNULL)
178
179         with open(env_variables_output_file, 'r') as file:
180             variables = {}
181             for line in file:
182                 variable, value = line.strip().split('=')
183                 variables[variable] = value
184
185         variables['Path'] =
186             ↪ f"${{os_env:PATH}};{self.version_path}\\thirdparty\\googletest\\bin;{self.vers
187
188         parent_dir_env_file = os.path.dirname(self.env_file)
189         if not os.path.exists(parent_dir_env_file):
190             os.makedirs(parent_dir_env_file)
191
192         with open(self.env_file, 'w') as file:
193             json.dump(variables, file, indent=2)
194
195         return
196
197     @logger_decorator.read_test_paths_from_fxotests_file_decorator
198     def read_test_paths_from_fxotests_file(self):
199         """ Reads lf file containing paths of tests. Writes the paths in the
200             ↪ test_paths list """
201
202         with open(self.all_htests_file, "r") as f:
203             list_tests = f.readlines()
204             list_tests = [ test.split()[1] for test in list_tests ]
205             list_tests = [ test.replace("~/", "") for test in list_tests ]
206             list_tests = [ path for path in list_tests if "fxo_test" in path ]

```

```

201         self.list_target_paths = list_tests
202         list_tests = [ os.path.dirname(test) for test in list_tests ]
203         list_tests = list(set(list_tests))
204         self.test_paths = list_tests
205
206     return
207
208
209 def get_exe_bat_file_path(self, path, target):
210     exe_file_path = ""
211     lf_file = os.path.join(self.version_path, path, "common", target +
212         ↪ ".lf")
213     with open(lf_file, 'r') as file:
214         for line in file:
215             words = line.split()
216             if len(words) > 0 and words[0] == "-output_name":
217                 exe_file_path = os.path.join(self.version_path, path, "exe",
218                     ↪ words[1] + ".exe")
219     if exe_file_path == "":
220         exe_file_path = os.path.join(self.version_path, path, "exe",
221             ↪ f"{target}.exe")
222
223     bat_file_path = exe_file_path + ".bat"
224
225     return exe_file_path, bat_file_path
226
227 @logger_decorator.build_target_decorator
228 def build_target(self, target, path):
229     os.chdir(os.path.join(self.version_path, path))
230     subprocess.run(f'mxbuild bat --skipComponentTests
231         ↪ --skipIntegrationTests --skipUnitTests @{target}', shell=True)
232
233 def is_filtered_target(self, target):
234
235     is_filtered = False
236     if self.exec_args.k is not None:
237         if not re.search(self.exec_args.k, target):
238             is_filtered = True
239     if self.exec_args.d is not None:
240         if re.search(self.exec_args.d, target):
241             is_filtered = True
242
243     return is_filtered

```

```

240
241 def add_exe_and_bat_path(self, exe_file_path, bat_file_path):
242
243     self.exe_paths.append(exe_file_path)
244     self.bat_paths.append(bat_file_path)
245
246     return
247
248
249 def build_targets_from_list_targets(self):
250
251     current_workdir = os.path.normpath(self.version_path)
252     for target_path in self.list_target_paths:
253         target_path_split = target_path.split("/")
254         path = os.sep.join(target_path_split[:-1])
255         target = target_path_split[-1]
256         exe_file_path, bat_file_path = self.get_exe_bat_file_path(path,
257             ↪ target)
258
259         if not self.is_filtered_target(target):
260             if (not os.path.isfile(bat_file_path) or not
261                 ↪ os.path.isfile(exe_file_path)):
262                 self.build_target(target, path)
263             if os.path.isfile(exe_file_path) and
264                 ↪ os.path.isfile(bat_file_path):
265                 self.add_exe_and_bat_path(exe_file_path, bat_file_path)
266                 logger_decorator.logger.info(f'The following target was kept:
267                     ↪ {target}')
268             else:
269                 logger_decorator.logger.error(f'Have tried to build the
270                     ↪ target: {target} but exe or bat file was not created")
271                 logger_decorator.logger.info(f'Ignoring the target:
272                     ↪ {target}')
273             else:
274                 logger_decorator.logger.info(f'The following target was
275                     ↪ filtered: {target}')
276
277     os.chdir( current_workdir )
278     return
279
280 @logger_decorator.read_args_from_bat_decorator
281 def read_args_from_bats(self):
282

```

```

276     for bat in self.bat_paths:
277         with open(bat, "r") as b:
278             lines = b.readlines()[:]
279             args = [lines[i + 1] for i in range(len(lines)) if
                ↪ lines[i].startswith("PUSHD")]
280             list_args = args[0].split()
281             list_args = list_args[1:-1]
282             self.env_args.append(list_args)
283
284     return
285
286
287 def write_json_dict_from_dicts(self):
288
289     self.dict_exe_args = {
290         "testMate.cpp.test.advancedExecutables" : [
291             {
292                 "pattern": pattern,
293                 "waitForBuildProcess": True,
294                 "envFile": self.env_name,
295                 "gtest": {
296                     "ignoreTestEnumerationStdErr": True,
297                     "prependTestRunningArgs": arg,
298                     "prependTestListingArgs": arg
299                 }
300             } for pattern, arg in zip(self.exe_paths, self.env_args)
301         ]
302     }
303
304     return
305
306
307
308 ↪ @logger_decorator.write_testmate_advanced_executables_in_vscode_settings_decorator
309 def write_testmate_advanced_executables_in_vscode_settings(self):
310
311     with open(self.vscode_settings_file, "r") as file:
312         data = json.load(file)
313
314     with open(self.vscode_settings_file, "w") as file:
315         data["testMate.cpp.test.advancedExecutables"] =
                ↪ self.dict_exe_args["testMate.cpp.test.advancedExecutables"]
        json.dump(data, file, indent=4)

```

```

316
317     return
318
319
320
321 def set_workingdir_to_version_root():
322
323     current_workdir = os.path.normpath(os.getcwd())
324     paths = current_workdir.split(os.sep)
325     os.chdir(os.sep.join([paths[0], paths[1]]))
326
327     return
328
329
330 def main(args):
331
332     fxoTestJsonWriter = FxoTestJsonWriter(args)
333     fxoTestJsonWriter.read_test_paths_from_fxotests_file()
334     fxoTestJsonWriter.build_targets_from_list_targets()
335     fxoTestJsonWriter.read_args_from_bats()
336     fxoTestJsonWriter.write_json_dict_from_dicts()
337
338     ↪ fxoTestJsonWriter.write_testmate_advanced_executables_in_vscode_settings()
339
340     return
341
342
343 if __name__ == "__main__":
344
345     parser = argparse.ArgumentParser(description="Script for loading tests in
346     ↪ TestMate")
347     parser.add_argument('-k', type=str, help='Keep argument')
348     parser.add_argument('-d', type=str, help='Discard argument')
349
350     args = parser.parse_args()
351
352     set_workingdir_to_version_root()
353
354     main(args)

```



## A.4 Interface de TestMate

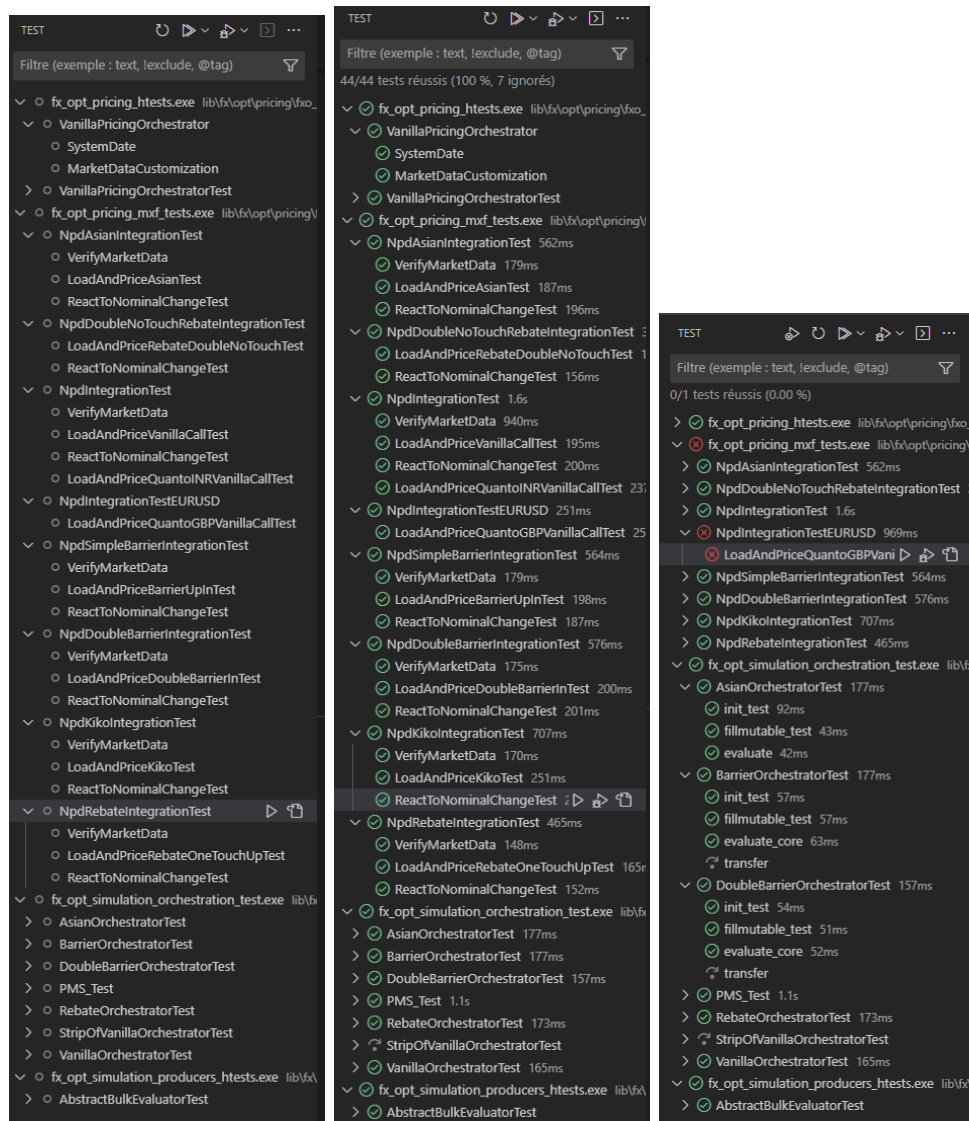


FIGURE A.5 – Exemple de l'interface de TestMate

## A.5 Couverture des tests



FIGURE A.6 – Comparaison de la couverture offline du dossier de l'équipe fxo avant mes tests (en haut) et après mes tests (en bas) (fichier de pricing)

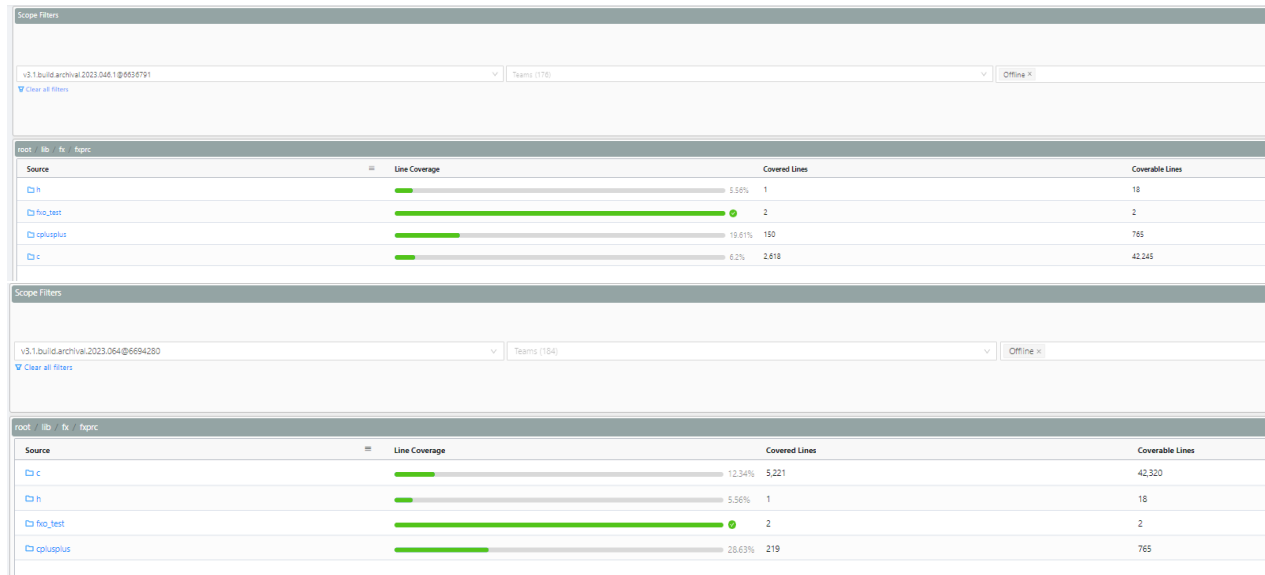


FIGURE A.7 – Comparaison de la couverture offline d'un dossier de pricing avant mes tests (en haut) et après mes tests (en bas)