INSA de Rennes
Quatrième année Informatique

Rapport de conception

Projet Value-at-Risk

Benjamin Bouguet - Damien Carduner
Paul Chaignon - Eric Chauty - Xavier Fraboulet
Clément Gautrais - Ulysse Goarant

Hamdi Raissi - Ivan Le Plumey
Quentin Giai Gianetto

Janvier 2014

Table des matières

Table des figures

Introduction

Dans les salles de marchés, les traders achètent et vendent des actions et obligations, pour le compte de grandes banques ou investisseurs toujours à la recherche d'un maximum de profits pour des pertes minimales. Il est intéressant pour les financiers de quantifier les risques de pertes, afin de prévoir des montants adéquats pour couvrir ces pertes éventuelles. En connaissant ces montants, leurs prévisions n'en seront que plus fiables et permettront peut-être d'éviter de mauvais placements.

C'est précisément ce que notre logiciel permettra de faire : calculer la Value-at-Risk par diverses méthodes statistiques. Néanmoins, un unique calcul de la Value-at-Risk n'est pas suffisant pour permettre de bonnes prévisions. Pour cette raison, notre programme aura l'avantage de proposer un ensemble de tests, comparaisons, tableaux récapitulatifs et même de la comparaison de méthodes statistiques par backtesting. Avec cette multitude d'outils, les financiers seront en mesure de travailler sur leurs portefeuilles aisément, au moyen d'outils de manipulations des actifs, mais aussi d'une interface intuitive et ergonomique, pensée pour ne montrer que l'essentiel et être facile d'utilisation.

Le rapport de spécifications fonctionnelles détaillait de manière très précise nos choix d'implémentation pour ce projet. Nous devons maintenant détailler nos choix techniques ainsi que nos décisions de modélisation. Dans un premier temps, nous allons rappeler les principales fonctionnalités de notre logiciel. Puis nous décrirons l'architecture globale du logiciel ainsi que les interactions entre les différents modules.

Dans un deuxième temps, nous décrirons à partir de diagrammes de classes et de séquences nos différents choix de conception. À partir desquels, nous détaillerons nos choix d'implémentations et d'interfaçage.

Chapitre 1

Rappel des spécifications

1.1 Cas d'utilisations

L'utilisateur de notre application aura besoin d'un ensemble d'outils pour gérer comme il le souhaite ses portefeuilles, effectuer des calculs et prendre ses décisions. Ceci peut se diviser en 3 grands pôles :

- la gestion de portefeuille ;
- le calcul de la Value-at-Risk ;
- le backtesting.

Ces derniers ont des dépendances respectives diverses. Par exemple, le backtesting utilise des fonctions du second pôle. En effet, son but est de comparer les différentes méthodes de calcul de la Value-at-Risk. Ces deux derniers modules devront stocker des résultats dans le portefeuille associé. Ces différentes interactions sont bien visibles à la figure 1.1.

Ces trois grands pôles sont détaillés dans les parties suivantes.

1.2 Gestion de portefeuille

La première étape dans toute gestion de portefeuille est la création. Pour cela, il faut commencer par importer des actifs stockés dans des fichiers de type CSV comme ceux fournis par Yahoo Finance par exemple. Ensuite l'utilisateur pourra sélectionner les données qu'il souhaite pour les ajouter dans un nouveau portefeuille. Il affectera alors une pondération (nombre d'actions s'il s'agit d'une action par exemple). La figure 1.2 illustre la description d'un actif au format CSV et provenant de Yahoo Finance. Nous utiliserons seulement les colonnes *Date* et *Close*.

1.3 Calcul de la Value-at-Risk

Le calcul de la Value-at-Risk est la fonctionnalité centrale de ce logiciel. Comme dit précédemment, elle s'effectue sur un portefeuille donné. De plus, le calcul de la Value-at-Risk se caractérise par la méthode utilisée (historique, Riskmetrics ou selon un modèle GARCH), un niveau de risque et un horizon de

temps. Il sera donc possible de calculer la Value-at-Risk selon les différentes méthodes et selon les paramètres précédemment décrits sur chacun des portefeuilles créés. Les résultats seront alors affichés dans un rapport. Néanmoins, proposer un simple calcul de Value-at-Risk n'est pas suffisant pour permettre une prise de décision. Il faut aussi s'assurer que la méthode de calcul de la Value-at-Risk est adaptée au comportement des actifs. Pour cela, l'utilisateur pourra effectuer du backtesting.

1.4 Backtesting

Le backtesting permet d'évaluer et de comparer les différentes méthodes de calcul de la Value-at-Risk. L'ensemble des résultats générés seront disponibles dans un rapport associé au portefeuille.

Lorsque l'utilisateur souhaite faire du backtesting, il doit choisir le portefeuille sur lequel l'effectuer, la ou les méthodes de calcul de la Value-at-Risk (historique, Riskmetrics ou GARCH), la période de test, le risque et l'horizon de temps.

1.5 Diagramme de composants

La figure 1.3 présente le diagramme des composants internes de notre logiciel. Nous avons réalisé les diagrammes de classes des différents composants séparément dans un premier temps. Les diagrammes de séquences nous ont ensuite permis d'établir les liens entre nos divers composants.

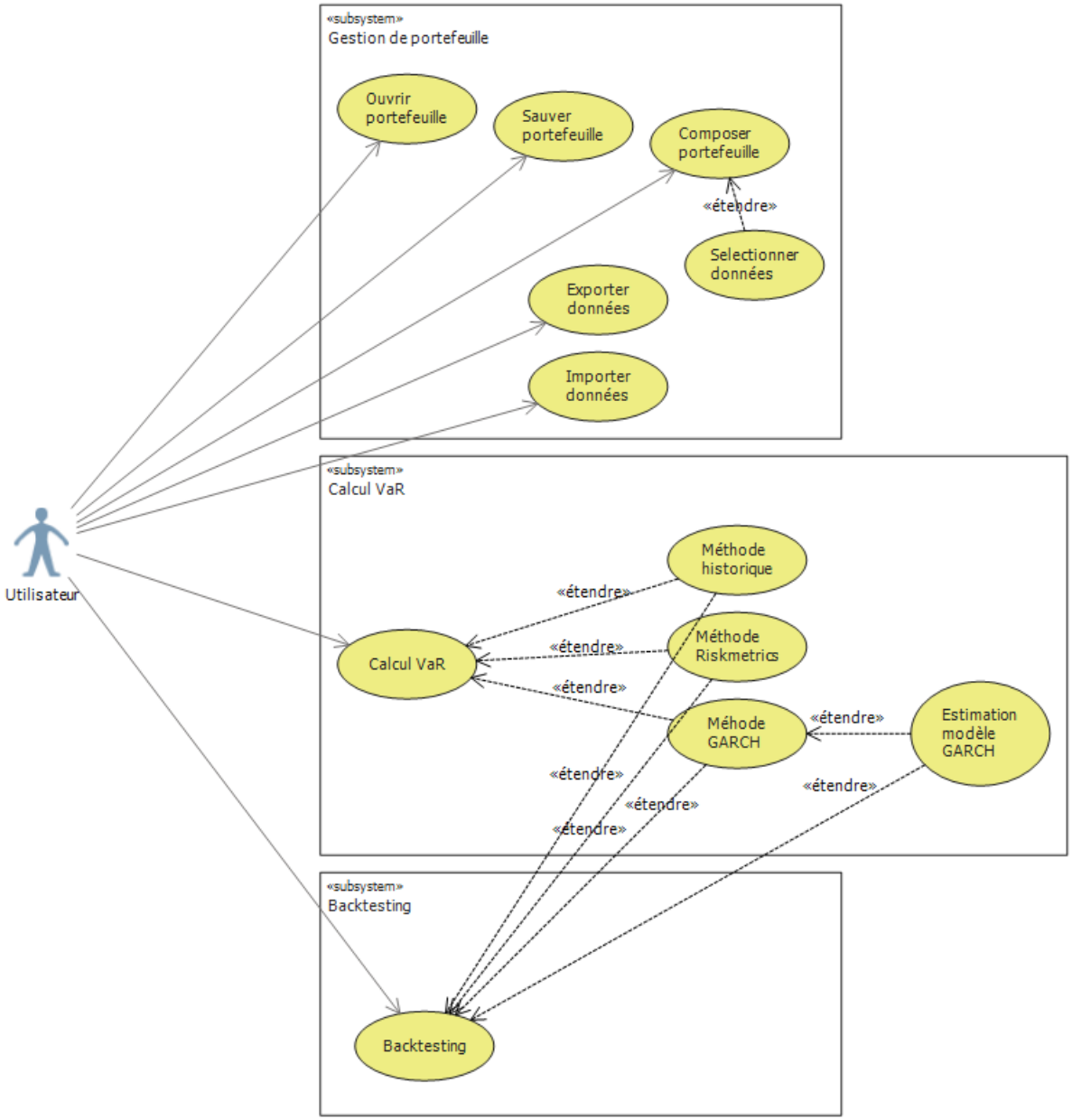

FIGURE 1.1 – Diagramme des cas d'utilisation

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-02-11,9338.80,9478.77,9338.01,9478.77,76248400,9478.77
2014-02-10,9331.71,9346.13,9280.18,9289.86,59599900,9289.86
```

FIGURE 1.2 – Extrait d'un actif au format CSV

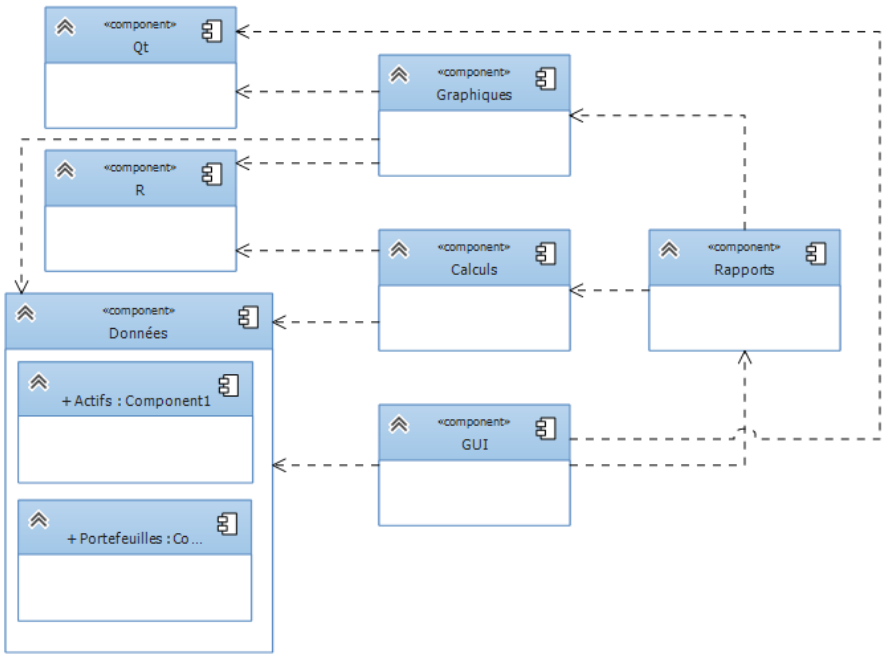

FIGURE 1.3 – Diagramme de composants

Chapitre 2

Conception

2.1 Modélisation de la base de données

2.1.1 Modélisation

La modélisation des données est organisée autour des portefeuilles. Un portefeuille est composé d'un nombre variable d'actifs avec diverses pondérations. Tous les rapports sont associés à un portefeuille.

De plus, un portefeuille sera amené à évoluer. Des actifs pourront par exemple être ajoutés, constituant ainsi un nouveau portefeuille dérivant du premier. Comme nous souhaitons conserver l'historique de ces évolutions nous garderons une référence vers le portefeuille d'origine dans le nouveau portefeuille.

Enfin, les valeurs des actifs seront conservées dans les fichiers CSV et XSL importés. La base de données contiendra donc des références vers ces fichiers. L'entité *Actifs* contient aussi l'intervalle de temps sur lequel est défini un actif. De même que pour les actifs, l'entité *Rapports* contiendra uniquement des références vers les fichiers générés.

La figure 2.1 présente le modèle entité-association de notre base de données. Nous en déduisons directement ses tables dans la figure 2.2. Il apparaît clairement que l'unique association ne pouvant être intégrée dans une entité est *Composé de*. Cette dernière constituera donc la table Pondérations pour faire le lien entre les portefeuilles et les actifs les constituant. Le champ *parent* de *Portefeuilles* sera nul pour représenter l'absence de parent.

2.1.2 Choix du Système de Gestion de Base de Données

Le nombre d'informations sauvegardées dans la base de données restera toujours raisonnable. La table dont la taille sera la plus importante sera vraisemblablement *Actifs*. Cependant le nombre d'actifs devraient rarement dépasser le millier. De plus, nous n'avons pas besoin de gérer d'accès concurrentiels à la base de données.

Nous pouvons donc utiliser le SGBD SQLite qui a l'avantage de ne pas nécessiter de serveur [1]. De plus, toute la base de données sera contenue dans un unique fichier et les accès en lecture comme en écriture seront beaucoup plus rapides qu'avec un SGBD traditionnel.

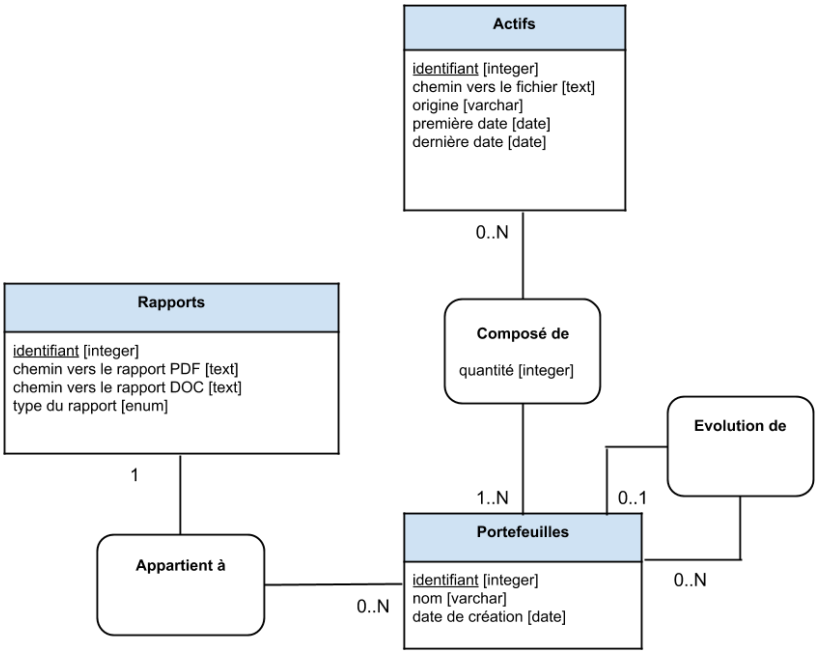

FIGURE 2.1 – Modèle entité-association

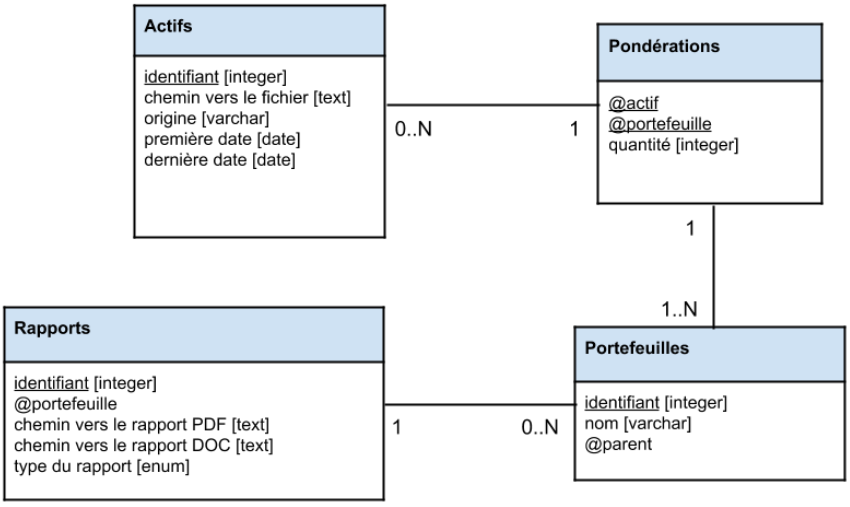

FIGURE 2.2 – Base de données

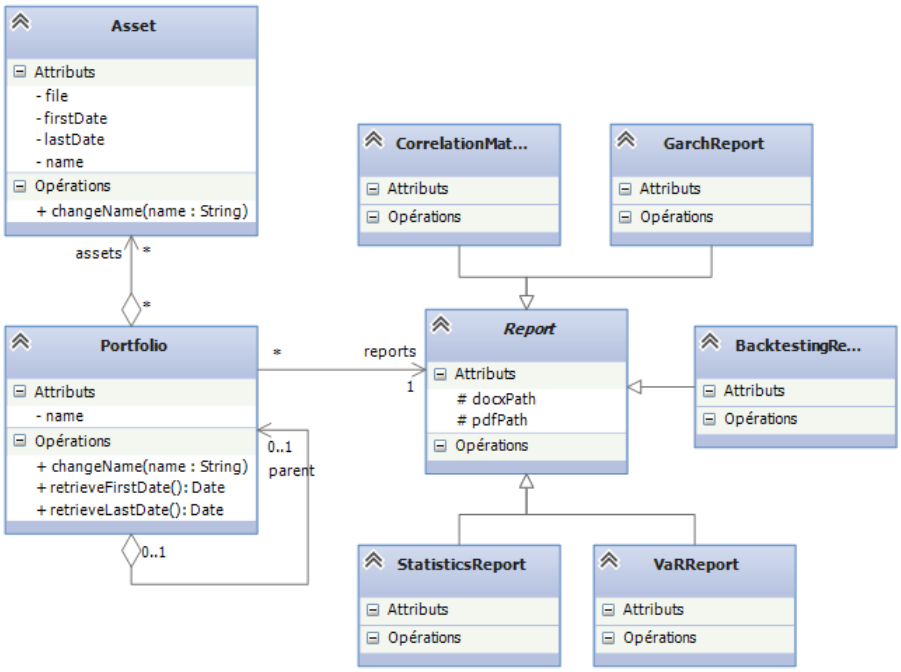

FIGURE 2.3 – Diagramme de classes des données

2.2 Diagrammes de classes

Cette partie détaille la modélisation pour les différents composants de notre logiciel. Nous commençons par la modélisation des données de base. Nous abordons ensuite le problème de l'instanciation des actifs. La suite des modélisations explique les composants nécessaires à la génération des rapports en passant par les interfaces avec quelques modules externes. Nous finissons avec le chargement de session et la fonctionnalité d'importation/exportation de portefeuilles.

2.2.1 Modélisation des portefeuilles

La modélisation sous forme de classes des données en figure 2.3 est analogue à celle pour la base de données. Un portefeuille contient des références vers ses rapports et ses actifs. Les divers attributs présents dans la base de données sont aussi présents ici.

De plus, un portefeuille dispose de deux méthodes (*retrieveFirstDate* et *retrieveLastDate*) pour calculer son intervalle de définition à partir de ceux de ses actifs.

2.2.2 Instanciation des actifs

Un même actif peut participer à la constitution de plusieurs portefeuilles. Il pourra donc être référencé dans plusieurs objets. Pour éviter toute duplication, nous utiliserons un patron de conception *Poids-Mouche* présenté en figure 2.4.

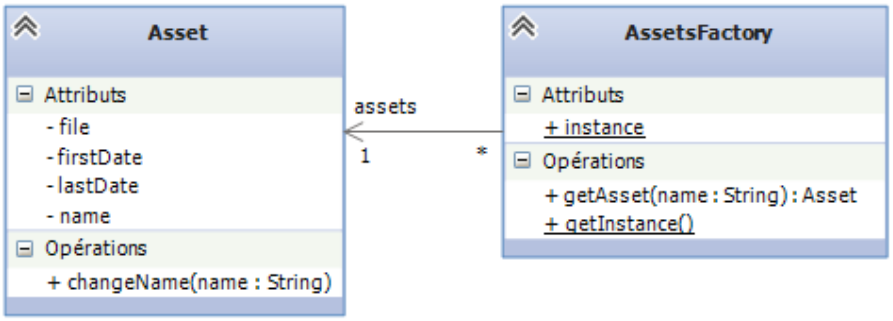

FIGURE 2.4 – Diagramme de classes *Poids-mouche* pour l'instanciation des actifs

Nous aurons donc une *Fabrique* d'actifs qui conservera une référence vers tous les actifs instanciés. Cette dernière est aussi un *Singleton*.

2.2.3 Calcul de la Value-at-Risk

Notre logiciel proposera trois méthodes [1] différentes pour calculer la Value-at-Risk. Nous utiliserons un patron de conception *Stratégie* (présenté en figure 2.5) pour modéliser ces méthodes.

Les trois stratégies prennent en entrée un portefeuille, un niveau de risque et un horizon de temps. Cependant, la méthode avec modélisation GARCH prend aussi en entrée le modèle GARCH. En effet, nous avons choisi d'externaliser l'estimation du modèle GARCH pour faciliter d'autres opérations (voir Backtesting avec modèle GARCH en 2.3.3). La stratégie avec modélisation GARCH pourra donc être instanciée avec différents modèles GARCH.

Toutes les stratégies seront instanciées pour un certain portefeuille et un certain niveau de risque. L'horizon de temps sera quant à lui passé en argument de la méthode de calcul, toujours pour faciliter le backtesting.

2.2.4 Génération des rapports

Les rapports seront générés aux formats PDF et DOCX. L'opération pourra donc prendre un peu de temps. Cette génération sera effectuée par une classe *ReportsFactory*.

Les différents types de rapports nécessitent des informations différentes pour leur génération. Nous utiliserons donc le patron de conception *Fabrique* présenté en figure 2.6. Les informations pour la génération seront passées à la fabrique lors de son instanciation.

2.2.5 Composants externes

Notre logiciel utilisera des composants externes pour répondre à certains besoins. Nous utiliserons la bibliothèque SQLite pour la base de données, des scripts R pour les calculs statistiques complexes et deux bibliothèques pour la génération des rapports.

1. La méthode historique, la méthode Riskmetrics et la méthode avec modélisation GARCH

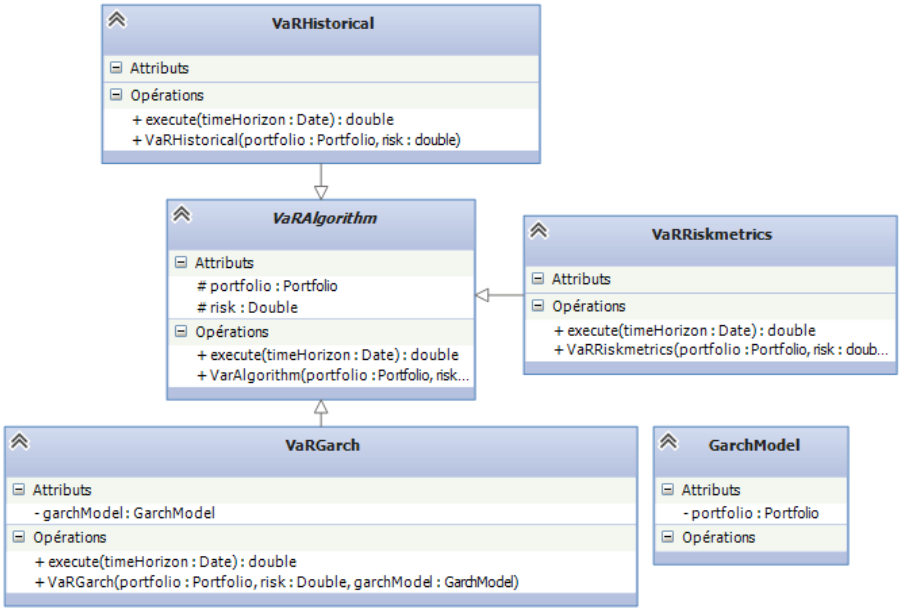

FIGURE 2.5 – Diagramme de classes *Stratégie* pour le calcul de la Value-at-Risk

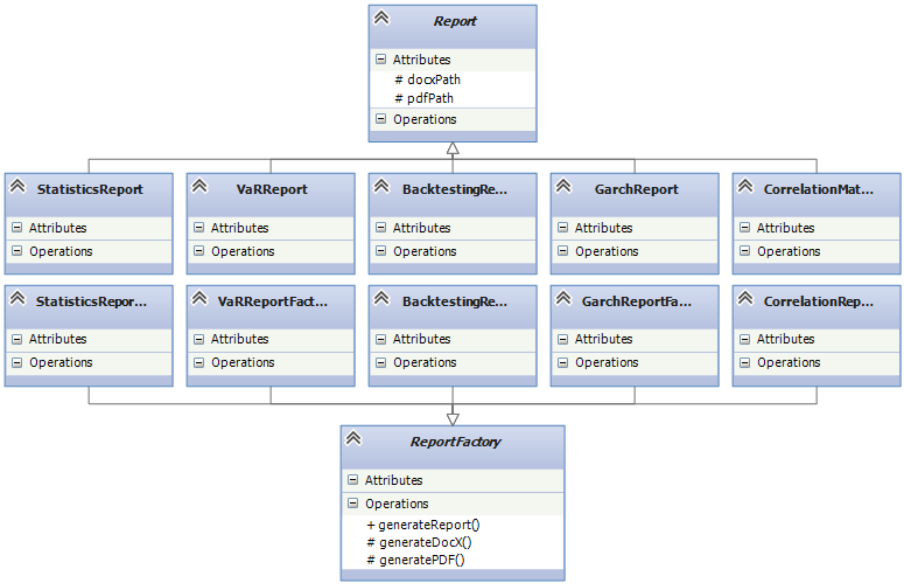

FIGURE 2.6 – Diagramme de classes pour la création des classes *Report*

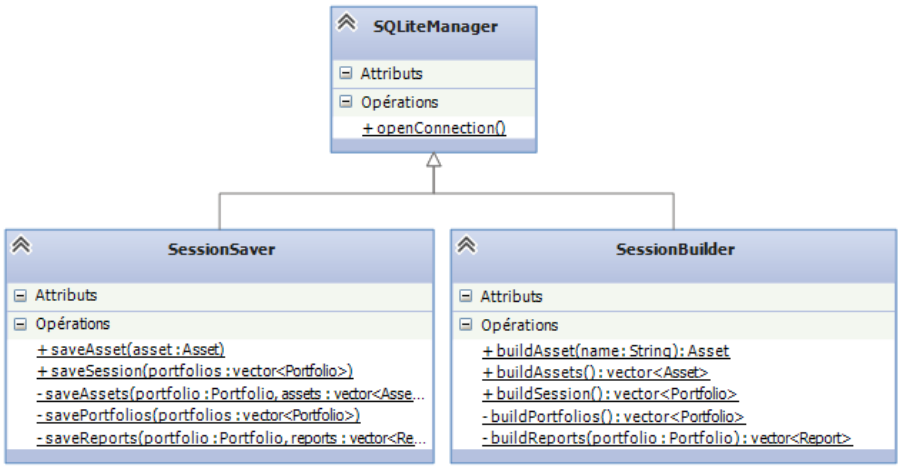

FIGURE 2.7 – Diagramme de classes pour la gestion de la session

Interface avec la base de données

Notre session sera conservée dans une base de données SQLite. Nous aurons quatre situations d'accès au fichier SQLite :

- La sauvegarde de la session courante [2].
- L'ouverture d'une session.
- Nous aurons aussi besoin d'accéder à la base de données lors de l'ajout d'un actif. Une méthode *saveAsset* permettra alors de sauvegarder ce dernier.
- Enfin, lors de son premier appel[3], le *Singleton AssetsFactory* chargera l'ensemble des actifs à l'aide de *buildAssets*. Si de nouveaux actifs sont créés par la suite *AssetsFactory* les chargera uniquement lorsqu'un portefeuille en aura besoin à l'aide de la méthode *buildAsset*.

La figure 2.7 présente les deux classes de sauvegarde et de restauration des sessions. Ces deux classes hériteront d'une classe *SQLiteManager* mettant à disposition des méthodes générales d'accès à la base de données.

Appel des scripts R

Certains calculs seront réalisés par des scripts R [2]. En effet ce langage est très bien adapté pour les calculs statistiques. À cet effet nous utiliserons la classe utilitaire décrite en figure 2.8. Elle permettra d'estimer le modèle GARCH et d'effectuer les tests de corrélation.

Les appels aux scripts R seront effectués en utilisant la bibliothèque *RInside* [3]. Cette dernière est bien documentée et permet simplement d'exécuter du code R dans du code C++.

2. La sauvegarde sera une action déclenchée par l'utilisateur uniquement.
3. *AssetsFactory.getInstance* sera appelé pour la première fois lors de la construction de la session par *SessionBuilder.buildSession*.

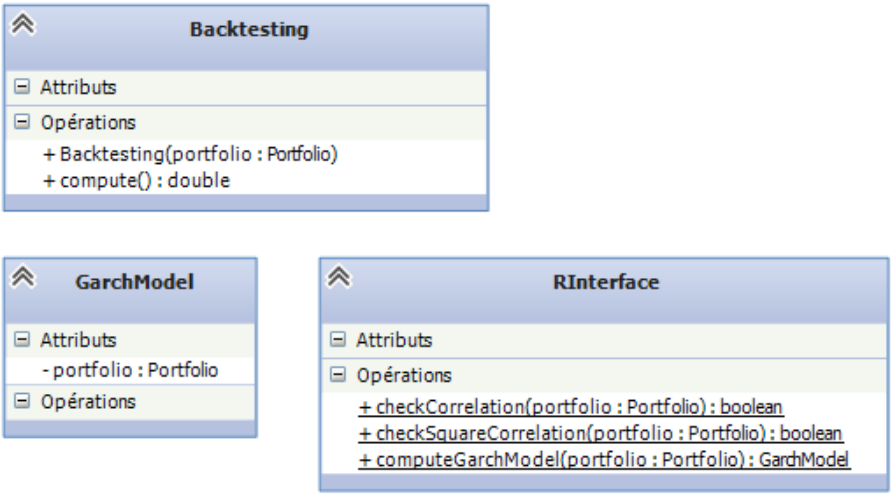

FIGURE 2.8 – Diagramme de classes de l'interfaces avec R

Génération des rapports

Pour la génération des rapports au format DOCX, nous nous appuierons sur la bibliothèque externe libopc [4]. Les rapports aux formats PDF seront ensuite générés depuis leurs homologues au format DOCX.

Pour anticiper les ajouts futurs d'autres formats de rapports, toutes les opérations de génération seront regroupées dans une interface. Les classes s'occupant de la génération des différents formats en hériteront. Nous aurons, par exemple, une méthode *addChart* dans l'interface pour ajouter un graphique dans un rapport.

Cette organisation, présentée à la figure 2.9, est celle d'un patron de conception *Monteur* où les appels des méthodes de construction sont effectués par les classes héritant de *ReportFactory*.

Génération des graphiques

Nous génèrerons les graphiques intégrés aux rapports à l'aide de QCustomPlot. La création de graphiques aurait aussi été possible avec R mais QCustomPlot nous permettra une plus grande maîtrise leur aspect. QCustomPlot est habituellement utilisé pour afficher les graphiques créés. Dans notre cas, nous nous contenterons de les sauvegarder sous forme de fichiers sans les afficher. Ainsi, ils pourront être utilisés lors de la compilation des rapports.

2.2.6 Chargement de la session

À l'ouverture de la session, tous les objets nécessaires seront chargés en mémoire vive depuis la base de données. Nous devrons donc instancier les objets portefeuilles, rapports et actifs. Nous utiliserons pour cela le patron de conception *Monteur* décrit au niveau de la classe *SessionBuilder* de la figure 2.7.

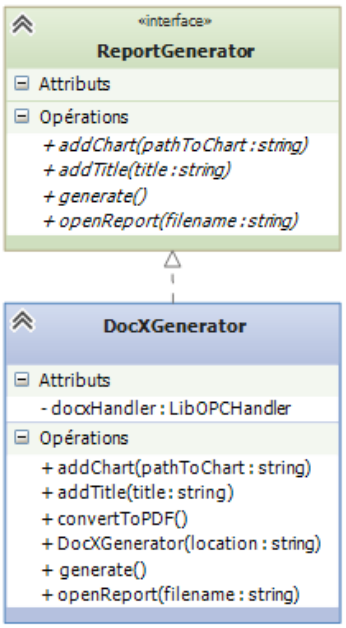

FIGURE 2.9 – Diagramme de classes interfaces avec les bibliothèques pour la génération des rapports

2.2.7 Importation et exportation de portefeuilles

Les fonctionnalités d'importation et d'exportation doivent permettre à un utilisateur d'exporter un ou plusieurs portefeuilles de sa session courante pour les importer dans une autre session. L'importation doit pouvoir être réalisée sur un autre ordinateur disposant du logiciel. De plus, l'utilisateur n'a pas besoin d'être capable de modifier le fichier d'export sans utiliser le logiciel.

Nous utiliserons pour cela un processus de sérialisation vers un fichier binaire. L'utilisateur pourra choisir d'exporter les rapports associés à un portefeuille ou seulement le portefeuille lui-même. Les actifs composant un portefeuille seront nécessairement exportés avec ce dernier. Nous aurons donc divers types d'objets à sérialiser. Tous ces objets disposeront de deux méthodes pour la sérialisation et la désérialisation. Nous utiliserons la classe *QVariant* de Qt à cet effet.

2.3 Études de cas

2.3.1 Création d'un portefeuille

Un portefeuille peut être créé à partir de zéro ou en modifiant un autre portefeuille. Dans les deux cas, l'utilisateur doit choisir les actifs à ajouter (et/ou à supprimer dans le second cas).

La figure 2.10 présente le diagramme de séquence de cette création.

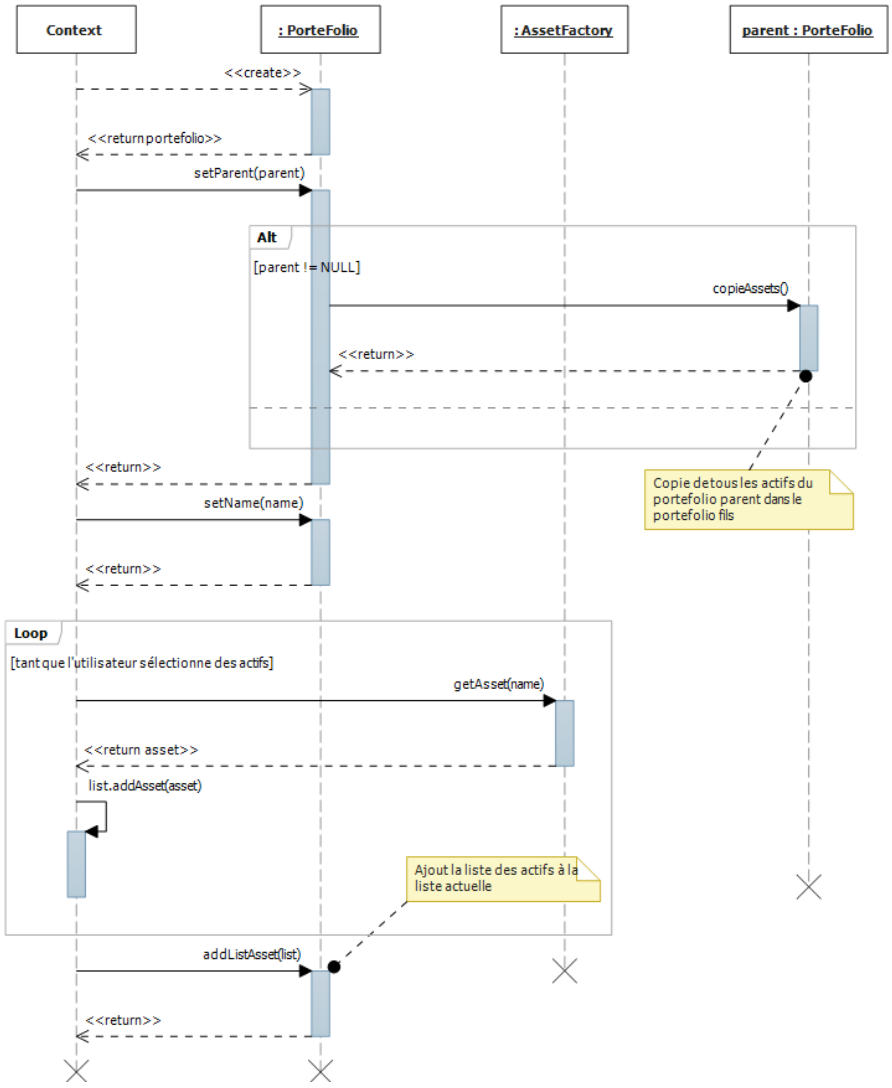

FIGURE 2.10 – Diagramme de séquence de la création d'un portefeuille

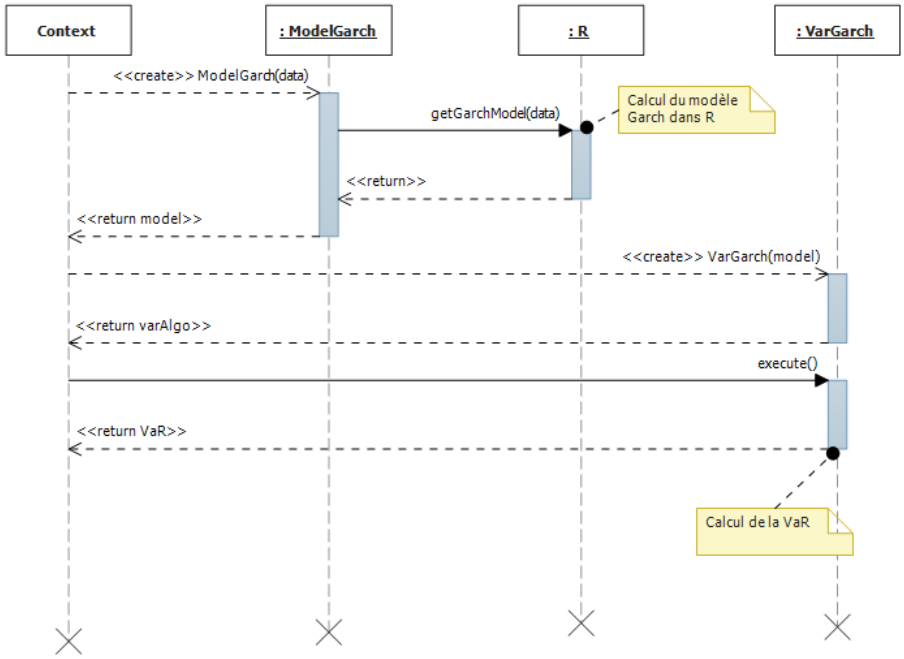

FIGURE 2.11 – Diagramme de séquence du calcul de la Value-at-Risk par GARCH

2.3.2 Calcul de la Value-at-Risk avec modèle GARCH

Le calcul de la Value-at-Risk avec un modèle GARCH nécessite dans un premier temps l'élaboration du modèle en lui-même. Ce dernier s'obtient par la méthode *getGarchModel* qui prend le portefeuille en paramètre. Cette modélisation est ensuite utilisée par la classe *VaRGarch* pour effectuer le calcul de la Value-at-Risk. La figure 2.11 détaille ces opérations.

2.3.3 Backtesting avec modèle GARCH

Un objet *Backtesting* est instancié avec les attributs suivants : la classe de calcul de la Value-at-Risk, le portefeuille, la période de temps sur laquelle les tests doivent s'effectuer, le risque associé à la Value-at-Risk et son horizon de temps. Cet objet possède une méthode *backtest* qui effectue le backtesting. Cette méthode fait un appel itératif au calcul de la Value-at-Risk selon la méthode correspondante sur chacun des jours de la période fixée. La figure 2.12 détaille ces opérations.

2.4 Interface graphique

Nous avons décidé d'utiliser l'architecture Modèle-Vue-Contrôleur (MVC) pour notre logiciel car étudiée en cours et très documentée. Cette architecture sépare les données, la présentation et les traitements facilitant ainsi toutes évolutions du logiciel. Qt permet de réaliser des logiciels avec ce type d'architecture

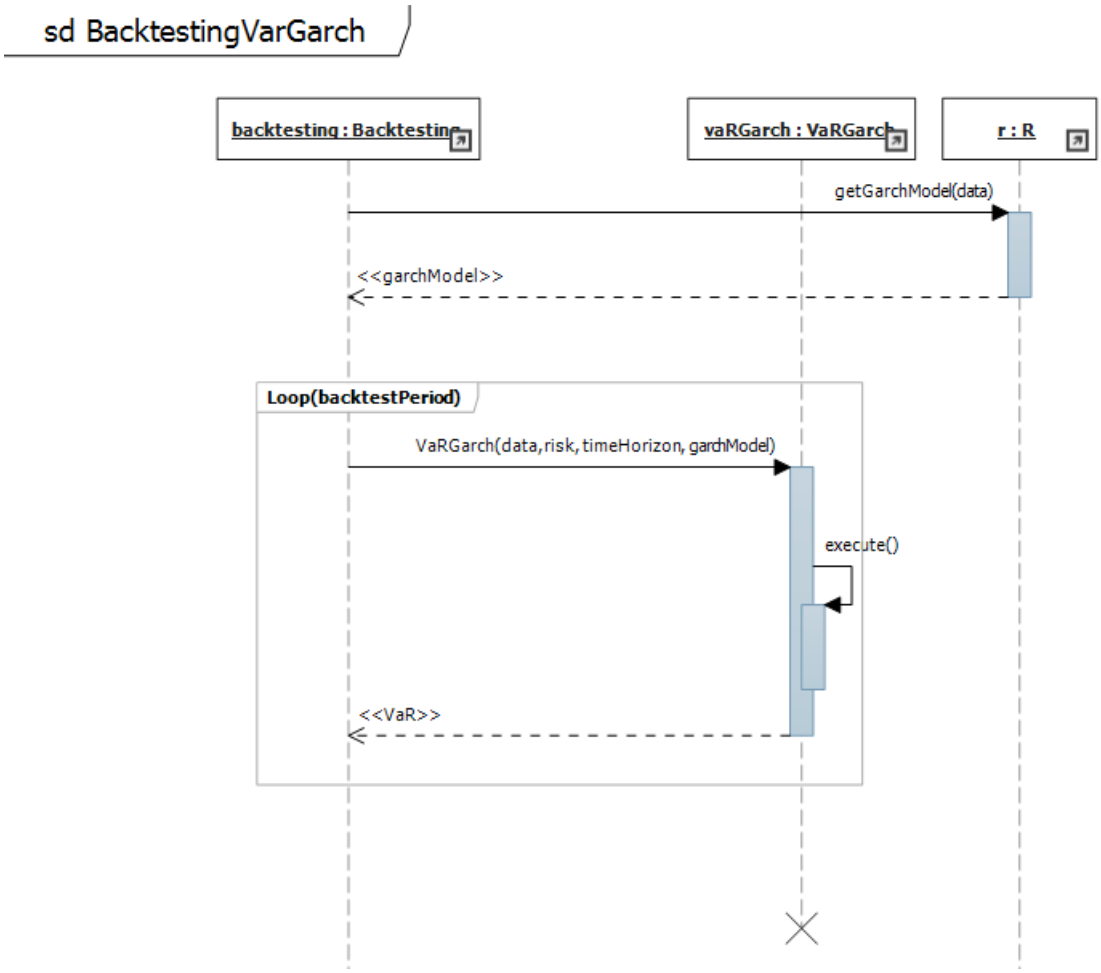

FIGURE 2.12 – Diagramme de séquence du Backtesting avec méthode GARCH

avec des outils comme QtDesigner pour concevoir graphiquement une fenêtre, donc sans écrire une seul ligne de code. La figure 2.13 présente un schéma de cette architecture.

QtDesigner permet de créer les vues de l'application qui sont ensuite enregistrées dans plusieurs fichiers. Il suffit ensuite de les charger dans le contrôleur. L'aspect graphique est donc complètement dissocié du code.

QtCore est la bibliothèque C++ de Qt qui contient des modèles de données pour les widgets Qt et des fonctionnalités de base qui ne concernent pas les interfaces graphiques. Elle sera utilisée dans le modèle de notre application pour les notifications de changement à la vue grâce au système de signaux/slots. Notre modèle contiendra les données (Actifs, Portefeuilles, Rapports) vues dans les paragraphes précédents. Ces données seront dans une bibliothèque statique (par exemple model.lib) pour une maintenance plus facile.

Nous aurons ainsi un projet C++/QtCore pour le modèle et un projet C++/Qt pour le contrôleur. Les fichiers avec les vues pourront être créés et stockés dans le projet du contrôleur.

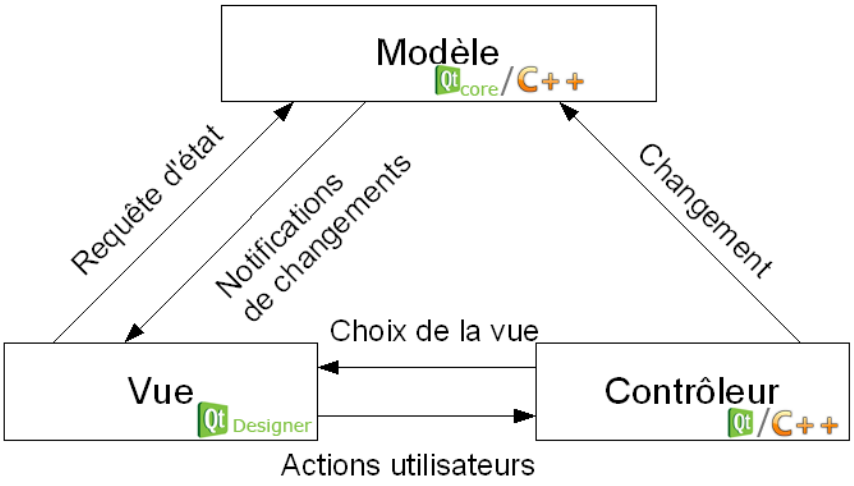

FIGURE 2.13 – Architecture MVC avec Qt

2.5 Tests du logiciel

2.5.1 Tests unitaires

Nous écrirons ces tests au fûr et à mesure du développement de notre logiciel, si possible avant même d'écrire le code à tester. Toute unité de code [4] non-triviale pouvant être testée avec des tests unitaires devra l'être. L'objectif est ici d'assurer la plus grande couverture de code pour éviter toute régression.

4. Dans notre contexte, une unité de code sera vraisemblablement une méthode comme nous ferons de la programmation orientée objet.

2.5.2 Tests empiriques

Estimation du modèle GARCH

L'estimation du modèle GARCH fait partie de nos algorithmes un peu délicat, notamment pour le tester. Nous utiliserons donc un test un peu particulier. Nous générerons tout d'abord, à l'aide de R, un ensemble de données à partir d'un modèle GARCH théorique. Nous aurons ainsi l'assurance que ces données suivent notre modèle GARCH. Nous pourrons donc ensuite estimer un modèle GARCH qui sera comparé au modèle théorique.

Calcul de la Value-at-Risk à partir d'un modèle GARCH

Le second algorithme délicat à tester est celui calculant la Value-at-Risk à partir d'une modélisation GARCH. De même, pour tester ce dernier nous effectuerons du backtesting avec des données générées depuis un modèle GARCH théorique. L'algorithme de backtesting [5] devrait donc retourner une valeur comprise dans un intervalle connu. En effet, cet intervalle est celui d'un théorème central limite avec pour paramètre le risque pris lors du calcul de la Value-at-Risk.

Ce deuxième test empirique dépend à la fois du calcul du modèle GARCH et de l'algorithme de Backtesting. Pour cette raison nous avons choisi de tester le premier calcul indépendamment. Le second calcul est relativement simple et pourra être l'objet d'un test unitaire [6].

2.5.3 Tests fonctionnels

Les tests fonctionnels permettront de valider chacun des cas d'utilisation classique de notre logiciel (voir le diagramme de cas d'utilisation en figure 1.1). Contrairement aux tests unitaires nous commencerons à les écrire vers la fin du projet, une fois que nous aurons suffisamment de modules opérationnels.

2.5.4 Tests de l'interface graphique

Les tests de l'interface graphique ne seront pas automatisés. Cette opération serait trop longue à effectuer. De plus, elle a peu d'intérêt d'un point de vue régression puisque l'interface graphique ne sera pas entièrement fonctionnelle avant la fin du projet. Enfin cette dernière sera amenée à changer suite aux retours du client. Avec des tests automatisés, toute modification de l'interface obligerait à modifier les tests.

Nous testerons donc cette partie manuellement au fûr et à mesure de son développement et entièrement à la toute fin du projet.

5. Voir 2.3.3 pour des explications sur le fonctionnement de l'algorithme.

6. En passant en paramètre une classe anonyme dérivée de *VarAlgorithm* dont nous définirons la valeur de retour.

Conclusion

Dans ce document, nous avons commencé par rappeler les fonctionnalités du logiciel définies lors de la phase de spécification à l'aide de diagrammes de cas d'utilisation. Parmi celles-ci, la gestion de portefeuilles, le calcul de la Value-at-Risk ainsi que la génération de rapports occupent une place centrale. Nous avons également présenté l'architecture générale du programme soulignant en particulier l'utilisation du module externe R pour les calculs avancés.

Nous avons ensuite détaillé la modélisation de la base de données nécessaire au stockage des informations importées ou générées afin que l'utilisateur retrouve d'une utilisation à l'autre ses portefeuilles et les rapports associés. Nous avons poursuivi avec la présentation de notre modélisation à l'aide de diagrammes de classes ainsi que quelques diagrammes de séquence afin de préciser les interactions les plus complexes entre les objets. En particulier, le calcul de la Value-at-Risk basée sur un modèle GARCH ou encore le backtesting ont bénéficié de cette dernière modélisation. Enfin nous avons abordé la modélisation de notre interface graphique ainsi que l'organisation de nos tests.

Lors de cette phase de conception nous avons été confrontés à un certain nombre de problèmes techniques auxquels nous avons dû répondre. Ce travail est nécessaire pour la phase suivante. En effet, nous allons maintenant pouvoir commencer le développement de notre logiciel. Grâce à cette étape préliminaire nous allons avancer plus rapidement et sereinement ; nous avons maintenant une vue précise du travail à effectuer.

Bibliographie

[1] Site officiel de la librarie sqlite. `http://www.sqlite.org/`. Consulté le 10 février 2014.

[2] Site officiel de r. `http://www.r-project.org/`. Consulté le 10 février 2014.

[3] Page officiel des classes rinside. `http://cran.r-project.org/web/packages/RInside/index.html`. Consulté le 10 février 2014.

[4] Page officiel de la librarie libopc sur codeplex. `http://libopc.codeplex.com/`. Consulté le 10 février 2014.

[5] Site officiel de la librarie libharu. `http://libharu.org/`. Consulté le 10 février 2014.