جامعة شعيب الدكالي
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵉ ⵛⵄⵉⴱ ⴷⴽⴰⵍⵉ
Université Chouaïb Doukkali

Royaume du Maroc
Ministère de l'Education Nationale, de la Formation Professionnelle
de l'Enseignement Supérieur et de la Recherche Scientifique
Université Chouaib Doukkali,

المدرسة العليا للتكنولوجيا
Ecole Supérieure de Technologie
Sidi Bennour - سيدي بنور

Rapport de Projet

Filière :

Génie Informatique

Système de Gestion de Stock Multi-Entrepôts Avec suivi des quantités

Présenté par :

M. ZAHI EL HOUSSINE
M. LOHANY AMINE
M. KHALIL SOUFIANE

Encadré par :

- **Encadrant Pédagogique**
  M. BADDI YOUSSEF

Année Universitaire 2025 – 2026

Remerciements

Nous tenons à remercier notre professeur référent pour son encadrement, ses conseils et ses orientations tout au long de la réalisation de ce projet.

Nous remercions également nos collègues et camarades pour leurs échanges constructifs, leurs relectures et leurs suggestions, ainsi que l'ensemble des ressources pédagogiques et documentaires qui nous ont permis de mener à bien ce travail.

Ces contributions ont joué un rôle important dans l'amélioration de la qualité technique et méthodologique du projet.

Sommaire

Introduction

1. Analyse des besoins

2. Conception

    2.1 Diagramme de classes

    2.2 Diagramme de séquence

    2.3 Cohérence et règles métier

3. Réalisation technique

    3.1 Langages et bibliothèques

    3.2 Architecture logicielle

    3.3 Persistance et migration

    3.4 Gestion des erreurs et tests

    3.5 Limitations techniques

4. Conclusion — Compétences acquises

➢ Introduction

Ce rapport présente la conception et la réalisation d'une application Java console dédiée à la gestion de stock réparti sur plusieurs entrepôts. Le contexte est celui d'organisations disposant de plusieurs sites de stockage — chaque site pouvant contenir des quantités différentes d'un même produit — et nécessitant une traçabilité complète des mouvements (entrées, sorties, transferts). La problématique centrale adressée est la cohérence des inventaires dans un environnement multi-entrepôts : comment représenter, valider et historiser les changements de quantité tout en gardant une logique simple à maintenir et à migrer vers une base de données relationnelle si besoin.

1. Analyse des besoins

À partir du code source et du cahier des charges présent dans le dépôt (**docs/CAHIER_DE_CHARGE.md**), les besoins fonctionnels identifiés sont :

- **Gestion des produits** : création, modification, suppression, consultation.

- **Gestion des entrepôts** : création, modification, suppression, consultation.

- **Enregistrement des mouvements de stock** : entrées et sorties avec date, quantité et référence.

- **Transfert inter-entrepôts** : opération atomique qui diminue la quantité dans l'entrepôt source et l'augmente dans l'entrepôt destination, tout en conservant un historique complet.

- **Consultation des états** : quantité par entrepôt et quantité totale par produit, calculées à partir de l'historique des mouvements.

- **Recherche et filtrage** : lister les mouvements par produit et/ou par plage de dates.

- **Robustesse métier** : empêcher les sorties ou transferts qui engendrent un stock négatif, et lever des exceptions métier dédiées.

Ces besoins sont implémentés dans les couches `**models**`, `**repository**`, `**services**` et `**exceptions**` de l'application. Les fichiers de données JSON (`**data/produits.json**`, `**data/entrepots.json**`, `**data/mouvements.json**`) servent de persistance légère pour l'application console.

2. Conception

La conception suit une séparation claire des responsabilités :

- `models` : objets métiers tels que `**Produit**`, `**Entrepot**` et `**MouvementStock**`.

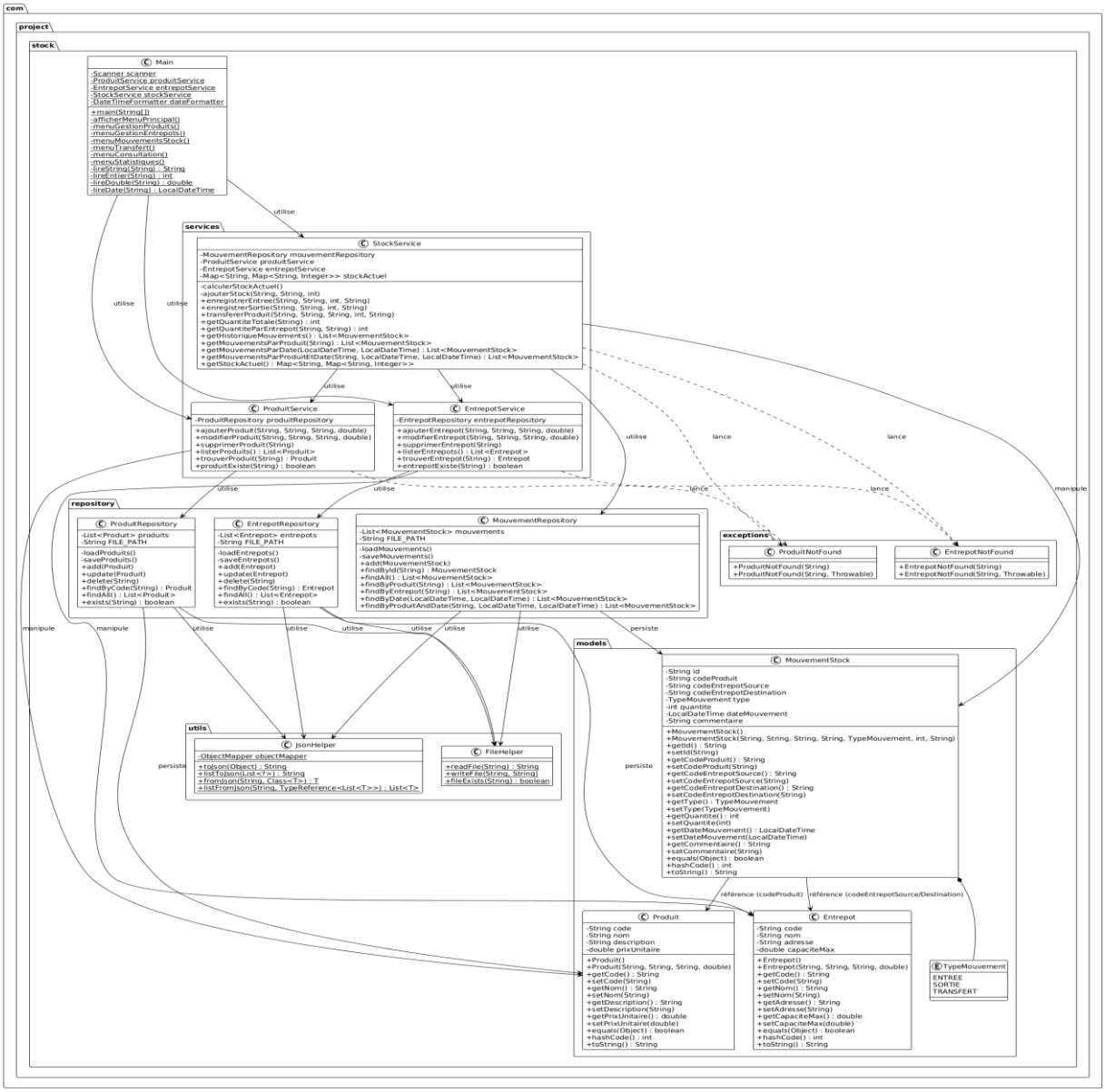
- `repository` : composants responsables de la persistance (**chargement/sauvegarde JSON**).

- `services` : couche métier qui applique les règles (**validation, calculs, orchestration des transferts**).

- `utils` : helpers pour lecture/écriture de fichiers et sérialisation JSON.

2.1 Diagramme de classes (conceptuel)

Le diagramme de classes (voir [**docs/diagramme_classes.png](docs/diagramme_classes.png)**) illustre les relations principales :



- `Produit` contient les attributs descriptifs (id, nom, SKU, etc.).

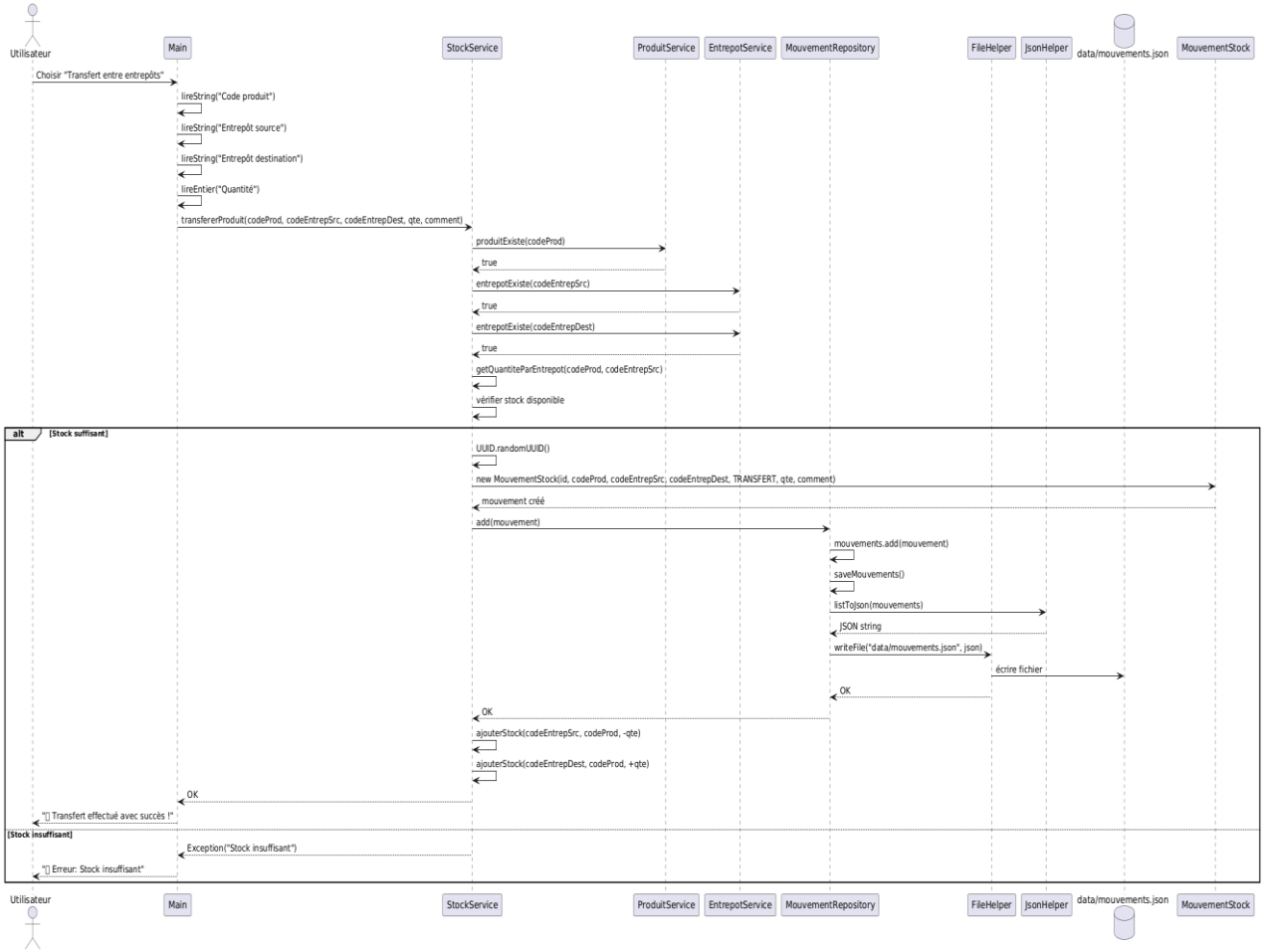
- `Entrepot` représente un site de stockage (id, nom, localisation).

- `MouvementStock` enregistre un événement immuable : type (`ENTRY`, `EXIT`, `TRANSFER`), `productId`, `warehouseId` (peut être null pour certains usages), `quantity`, `timestamp`, et éventuellement une `reference` ou `note`.

La logique de stock n'est pas stockée comme un champ mutable global ; au contraire, l'état disponible est dérivé en agrégant les `MouvementStock` pour un `productId` par `warehouseId`. Cette approche garantit une traçabilité complète et évite les divergences typiques des compteurs mutables.

2.2 Diagramme de séquence (exemple de transfert)

Le diagramme de séquence (voir docs/diagramme_sequence.png) montre l'ordre d'appel lors d'un transfert :



1. L'utilisateur demande un transfert via `Main` (interface console).

2. `StockService` orchestre l'opération : demande au `ProduitRepository` et `EntrepotRepository` de valider l'existence des entités.

3. `StockService` calcule le stock disponible en agrégeant `MouvementRepository` (somme des mouvements jusqu'à présent).

4. Si le stock source est suffisant, `StockService` crée deux enregistrements `MouvementStock` : un `EXIT` pour l'entrepôt source, et un `ENTRY` pour l'entrepôt destination (liés par une référence commune si nécessaire).

5. Les deux mouvements sont persistés via `MouvementRepository` et écrits dans `data/mouvements.json`.

Cette séquence garantit que l'opération de transfert laisse une trace complète et que la lecture ultérieure (recalcule de stock par agrégation) reflète correctement l'état.

2.3 Cohérence et règles métier

- **Validation préalable :** existence des identifiants et quantité positive.

- **Vérification anti-sur-soustraction :** refus d'un `EXIT` ou d'un transfert si la quantité disponible est insuffisante.

- **Atomicité logique** : même si la persistance JSON n'offre pas de transaction multi- fichier, l'application s'efforce d'écrire les deux mouvements consécutifs et d'indiquer toute erreur via des exceptions métier (`EntrepotNotFound`, `ProduitNotFound`, etc.). Pour un passage en base relationnelle, on pourra placer l'opération dans une transaction JDBC/JPA pour garantir l'atomicité physique.

3. Réalisation technique

3.1 Langages et bibliothèques

- Java 11+ comme langage principal.

- Maven pour la construction et la gestion des dépendances.

- Jackson pour la sérialisation/désérialisation JSON (persistance légère dans `data/`).

3.2. Architecture logicielle

Le projet suit un style proche du pattern MVC (ou plutôt une séparation en couches) :

- **Présentation / UI :** **`Main.java`** fournit une interface console pour l'utilisateur (menus, saisie).

- **Contrôleurs / Services :** classes de la couche `services` implémentent la logique métier et orchestent les appels aux **`repository`.**

- **Modèle / Persistence :** **`models`** et **`repository`** gèrent la représentation des données et la lecture/écriture sur disque.

3.3. Persistance et migration

La persistance actuelle est basée sur des fichiers JSON afin de faciliter la démonstration et les tests locaux. Les dépôts (`repository`) encapsulent le format de stockage, ce qui rend la migration vers une base relationnelle **(MySQL, PostgreSQL)** ou un moteur NoSQL relativement straightforward : il suffit d'implémenter une nouvelle version des repositories (**JDBC / JPA / Spring Data**) sans modifier la couche `services`.

Un schéma SQL minimal a été fourni dans le README pour guider une migration manuelle.

3.4. Gestion des erreurs et tests

Des exceptions métier dédiées **(`EntrepotNotFound`, `ProduitNotFound`)** sont utilisées pour séparer les erreurs attendues (données manquantes ou contraintes métier) des erreurs système (I/O, sérialisation). Le découpage facilite l'écriture de tests unitaires ciblés pour la couche `services` et des tests d'intégration pour les **`repository`.**

3.5. Limitations techniques

- **Persistance par fichier :** pas de transaction ACID native, risque de corruption en cas d'interruption lors d'écritures concurrentes.

- **Interface console :** adaptée pour démonstration, mais pas pour une exploitation multi-utilisateur. Une API REST ou une GUI seraient nécessaires pour usage en production.

4. Conclusion — Compétences acquises

Le projet illustre des compétences concrètes en ingénierie logicielle : modélisation objet, séparation des responsabilités, conception centrée sur la traçabilité des événements métier et préparation à la migration vers des solutions de persistance plus robustes. Sur le plan technique, l'auteur démontre la maîtrise de Java moderne, de la sérialisation JSON avec Jackson, et de patterns d'architecture pragmatiques **(services + repository).** Sur le plan méthodologique, la conception par événements (historique immuable des mouvements) montre une approche robuste pour garantir l'auditabilité et la reconstruction d'état.

Pour aller plus loin, les évolutions recommandées comprennent :

- Implémentation d'une couche de persistance relationnelle (JDBC/JPA) avec transactions pour garantir l'atomicité des transferts.

- Ajout de tests unitaires et d'intégration automatisés.

- Mise en place d'une API REST et d'un client web léger pour usage multi-utilisateur.

Références dans le dépôt :

- Spécification et cahier des charges : docs/CAHIER_DE_CHARGE.md

- Diagrammes UML : docs/diagramme_classes.png, docs/diagramme_sequence.png

Fichier(s) source principaux : `src/main/java/com/project/stock/` (models, services, repository, utils, exceptions, Main).