ChatGPT

Architecture Clean pour un Gestionnaire de Stock de Produits

Contexte et Objectifs du Projet

Dans ce projet, l'objectif est de développer un programme de gestion de stock modulaire et évolutif, en appliquant les meilleures pratiques d'architecture logicielle. Le système doit constituer une base robuste, facilement **adaptable à différentes interfaces utilisateur**, tout en étant **testable en mode terminal** (sans interface graphique). Cet impératif rejoint les principes de la **Clean Architecture**, qui prône une séparation stricte entre la logique métier centrale et les détails d'implémentation extérieurs (UI, base de données, frameworks, etc.) [1] [2]. En d'autres termes, on cherche à **découpler le noyau métier** du programme (gestion des produits et du stock) de toute dépendance vis-à-vis de l'interface visuelle ou du stockage des données, afin de garantir une meilleure pérennité du code et une grande facilité de test et de maintenance.

Exigence de qualité : Cette architecture doit être suffisamment solide et professionnelle pour surpasser les solutions concurrentes. Elle s'appuiera sur des principes reconnus (SOLID, Clean Architecture) et sur des patrons de conception appropriés. L'idée est de **maximiser la lisibilité, la maintenabilité et la testabilité** du code, ce qui se traduira par un produit final de haute qualité, bien documenté et prêt à être étendu dans le futur. Comme le souligne la Clean Architecture, la priorité est de garder intactes les règles métier au centre, indépendantes des détails technologiques extérieurs [3] [4]. Cela permettra de changer de technologie (par exemple, passer d'une interface console à une interface graphique riche) sans impacter la logique de gestion de stock.

Principes d'Architecture Adoptés

- **Séparation des responsabilités (Single Responsibility)** : Chaque composant du système aura un rôle bien défini et unique. Par exemple, les classes du domaine modélisent les données métier, les cas d'usage gèrent les opérations métier, les composants d'interface s'occupent de l'IHM, etc. Cette séparation claire en couches distinctes facilite la compréhension et la maintenance du code [5].

- **Indépendance vis-à-vis de l'UI et de la base de données** : La logique métier du stock ne dépendra ni d'un framework d'interface graphique, ni d'une technologie de stockage particulière. La Clean Architecture prescrit que l'interface utilisateur et la persistance sont des *"détails d'implémentation"* relégués en périphérie du système [6] [7]. Concrètement, on doit pouvoir **tester toutes les fonctionnalités métier sans avoir d'UI ni de BD connectée** [8]. Une interface console permettra de valider le système, et on pourra ultérieurement brancher une GUI sans modifier le cœur du code.

- **Testabilité et flexibilité** : En respectant ces principes, on obtient un système facilement testable et évolutif. Les composants métier peuvent être instanciés et testés isolément (par exemple via des tests unitaires ou un pilote en console) puisque les dépendances externes sont soit absentes, soit simulées via des interfaces [9] [10]. Cette architecture modulaire facilite aussi l'extension des fonctionnalités : ajouter un nouveau cas d'usage ou changer le mode de stockage

n'affectera pas les autres couches, réduisant le risque de régression [11]. On obtient ainsi un **code durable et propre**, conforme à l'esprit *"code à l'épreuve du temps"* de la Clean Architecture [12].

En résumé, nous allons structurer l'application en **couches concentriques** selon la Clean Architecture : le centre contient le domaine métier (entités et règles métier générales), autour se trouvent les cas d'usage spécifiques, puis les adaptateurs d'interface, et enfin les éléments externes (UI, base de données, frameworks) en périphérie [1] [13]. La règle de dépendance sera respectée : les dépendances pointent vers l'intérieur, c'est-à-dire que les couches externes dépendent des couches internes (appel de services, utilisation d'interfaces définies dans le noyau) et non l'inverse [4].

Structure en Couches (Clean Architecture)

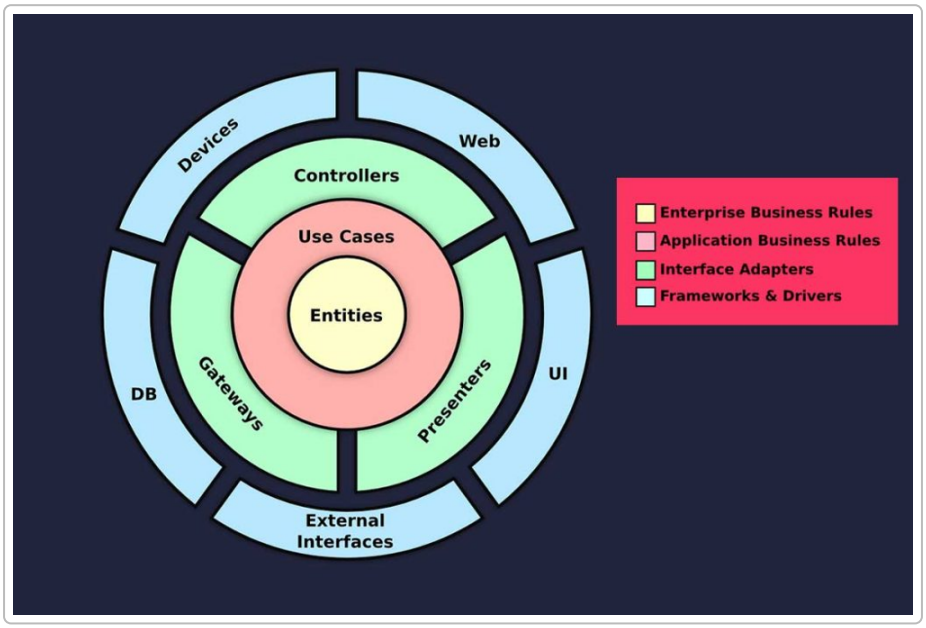

Schéma illustrant l'architecture en couches concentriques inspirée de la Clean Architecture (cœur Entités, couche Cas d'utilisation, couche Adaptateurs d'interface, périphérie Frameworks & Drivers).

La solution est organisée en **quatre couches principales**, chacune remplissant un rôle précis [14] :

1. **Couche Domaine – Entités métier** : Ce sont les objets du domaine de la gestion de stock. Ils représentent les données et les règles de gestion de base, sans dépendre de technologies externes [15]. Dans notre cas, il s'agit principalement de la classe abstraite `Product` et de ses sous-classes concrètes (`SimpleProduct`, `ContainerProduct`). Ces entités encapsulent les attributs des produits (identifiant, nom, prix, stock…) et garantissent des invariants (par exemple, un prix ou un stock ne peut être négatif, vérifié dans le constructeur). Elles portent le « métier » fondamental : par exemple, un `ContainerProduct` représente un produit qui **contient un autre produit** en quantité donnée (p. ex. un pack de X unités d'un produit) et doit respecter des contraintes (produit contenu non nul, quantité positive). Grâce à l'héritage `Product`, un conteneur peut être manipulé polymorphiquement comme un produit ordinaire – ce qui est justement l'idée du patron **Composite** appliqué aux produits. En effet, on peut voir un conteneur comme une composition de produit(s) : *« Une boîte peut contenir plusieurs produits ainsi que d'autres boîtes plus petites… »* [16]. Cette conception rend possible une structure arborescente de produits (un conteneur pouvant lui-même contenir un autre conteneur, etc.), bien que dans

notre implémentation chaque `ContainerProduct` se limite à un seul type de produit contenu à la fois (avec une quantité fixe). L'important est que la couche domaine se concentre sur **quoi** est un produit et quelles règles de base il respecte, sans aucune logique d'interface ou de persistance.

2. **Couche Application – Cas d'usage** : Autour des entités se trouve la logique applicative spécifique au système de stock [17]. Cette couche définit **ce que l'utilisateur peut faire** avec le système, en orchestrant les entités du domaine pour réaliser des opérations métiers complètes. Concrètement, on y placera les **services ou interactors** qui gèrent le stock. Par exemple, on pourra définir une classe `InventoryService` (ou plusieurs use-case classes séparées) contenant des méthodes pour :

3. *Ajouter un nouveau produit* dans le stock (en créant l'entité Product appropriée – simple ou conteneur – et en la passant à la couche de persistance).
4. *Consulter la liste des produits* disponibles, ou rechercher un produit par critère (par ID, par nom, etc.).
5. *Réapprovisionner* le stock d'un produit (augmenter la quantité en stock).
6. *Vendre un produit* à un client (diminuer le stock d'un certain nombre d'unités, avec les contrôles nécessaires).
7. *Vendre un produit conteneur* : ce cas particulier implique de décrémenter non seulement le stock du conteneur vendu, mais possiblement aussi le stock du produit contenu. Par exemple, vendre 1 pack de 6 bouteilles pourrait réduire le stock du pack **et** soustraire 6 unités du produit « bouteille » de base. La décision de mettre à jour le stock du contenu peut dépendre des règles métier (souhaite-t-on suivre individuellement le stock des composants ?). **C'est dans cette couche que de telles règles seront implémentées**, en manipulant les objets `ContainerProduct` et `SimpleProduct` de manière coordonnée.

La couche application ne **contient pas de logique d'interface** (pas d'affichage ni de lecture d'entrée) et ne s'occupe pas du stockage durable des données, mais elle peut faire appel à des **interfaces** pour ces besoins. Elle utilise typiquement un **répertoire (repository)** abstrait pour récupérer ou persister les entités du domaine. En encapsulant les règles métier de plus haut niveau (transactions de stock, règles de vente, etc.), elle assure que toutes les actions métier respectent les invariants du domaine. Chaque cas d'usage étant centralisé ici, il devient facile à tester isolément (en simulant les réponses du repository par exemple).

1. **Couche Interface Utilisateur – Adaptateurs** : Cette couche comprend les éléments qui font le lien entre les cas d'usage et le monde extérieur [18]. On y retrouve notamment les **contrôleurs** (controllers) et éventuellement des **présentateurs** ou **view models**. Dans notre gestionnaire de stock, cela va se traduire par une interface en ligne de commande (CLI) pour les tests, et potentiellement une interface graphique ultérieure. Par exemple, on peut avoir un module `ConsoleUI` qui affiche un menu texte et permet à l'utilisateur de saisir des choix (ajouter un produit, lister le stock, effectuer une vente, etc.). Ce contrôleur console va **interpréter les entrées utilisateur** (par ex. un choix de menu, ou un identifiant de produit saisi) et appeler la méthode correspondante du service de la couche application (cas d'usage). Inversement, il prendra les résultats renvoyés (par ex. la liste des produits ou le succès d'une opération) et les **adaptera au format de sortie** approprié (affichage texte dans la console). De même, si on implémente plus tard une interface graphique (desktop ou web), on développera de nouveaux adaptateurs (par ex. un contrôleur Swing/JavaFX ou un contrôleur web) qui joueront le même rôle : traduire les actions de l'utilisateur en appels aux use cases, puis présenter les données métiers dans la vue. L'important est que la logique métier ne "sait" rien de la façon dont elle est affichée ou des mécanismes d'input utilisateur [19]. Grâce à cela, **l'UI peut changer sans**

impacter le reste [20]. Techniquement, on veillera à ce que les adaptateurs ne manipulent le domaine qu'au travers d'interfaces ou de DTO : par exemple, le contrôleur pourrait demander au service application une liste d'objets métier et la convertir en texte formatté, ou construire un petit objet intermédiaire prêt pour l'affichage (c'est le pattern du *View Model* évoqué en Clean Architecture [21] [22]). Pour un mode console, cette couche reste simple (formatage de texte), mais elle est bien distincte de la couche application afin de préserver la testabilité et l'évolutivité.

2. **Couche Infrastructure – Persistance et Détails techniques** : La couche la plus externe contient les implémentations concrètes des services techniques nécessaires au fonctionnement du système [3] [23]. Dans le contexte du gestionnaire de stock, le principal élément d'infrastructure est le **système de stockage des données**. On définira par exemple une interface `ProductRepository` (dans la couche application ou domaine) décrivant les opérations nécessaires (ajout, recherche, mise à jour de produits). **L'infrastructure fournira une implémentation concrète** de cette interface – pour démarrer, ce sera une **base de données en mémoire** (par exemple une simple collection `List` ou `Map` stockant les produits). Cette implémentation permettra de tester l'application en mode console sans configuration lourde. Par la suite, on pourra développer une implémentation utilisant une base de données réelle (SQL, NoSQL) ou un fichier, **sans changer le code métier** qui utilise le repository. Ce découplage est rendu possible grâce au principe d'inversion de dépendance : la couche application définit une interface, et la couche infrastructure "branche" son implémentation via injection de dépendance. Ainsi, *« l'accès à la base de données est abstrait et isolé dans la couche d'infrastructure »*, le code métier ne dépend pas d'une technologie de BD spécifique, ce qui permet de changer de système de persistance facilement [7]. Outre la persistance, la couche infrastructure pourrait contenir d'autres détails techniques si nécessaire, par exemple un module d'envoi de notifications, des adaptateurs pour des services externes, etc., toujours implémentés de manière à ne pas impacter les couches supérieures.

Avec cette organisation en couches, **toute modification d'une couche externe n'affectera pas les couches internes** [4]. Par exemple, remplacer la console par une GUI n'exigera aucune modification des cas d'usage ou des entités. De même, passer d'un stockage en mémoire à une base de données ne changera rien pour le domaine ni pour la logique applicative, hormis la configuration de l'injection du nouveau repository. Cette isolation est la garantie d'une architecture *propre* et modulaire.

Avantages de cette Architecture

En adoptant cette architecture, le programme de gestion de stock bénéficiera de multiples avantages concrets :

- **Testabilité accrue** : On peut tester la logique de stock sans lancer d'interface graphique ni de base de données. Par exemple, des tests unitaires peuvent vérifier les règles de vente (y compris les cas des conteneurs) en simulant un repository en mémoire. La séparation nette des préoccupations permet de vérifier chaque couche isolément [24]. Lors de l'exécution en terminal, on aura un proof-of-concept fonctionnel sans dépendances externes, ce qui correspond exactement au critère *"système testable sans UI, BD..."* de la Clean Architecture [8].

- **Évolutivité et flexibilité** : La structure en couches rend le code **facile à faire évoluer**. Ajouter une nouvelle fonctionnalité (par ex. une fonctionnalité de gestion des fournisseurs ou une alerte sur stock minimum) se fera en créant de nouveaux use cases ou en étendant le domaine, sans refondre l'ensemble. De même, on pourrait réutiliser le cœur métier pour une application mobile ou web à l'avenir, en n'ayant qu'à écrire de nouveaux adaptateurs d'interface [25]. La logique

métier reste **indépendante des technologies**, donc pérenne malgré les changements de frameworks ou de bibliothèques [26].

- **Maintenabilité et lisibilité** : En divisant clairement les responsabilités, le code est plus lisible et bien organisé. Chaque développeur (ou chaque partie du projet) peut se concentrer sur une couche à la fois. La maintenance s'en trouve facilitée, car comprendre un module ne nécessite pas de démêler des enchevêtrements entre l'UI, la logique et la BD – ces éléments sont découplés [27]. De plus, l'utilisation de standards reconnus (patterns de conception, principes SOLID, documentation UML) rend le projet **professionnel** et compréhensible par d'autres développeurs. Ce soin apporté à l'architecture et à la qualité du code est un critère de distinction qui contribuera sans aucun doute à obtenir une excellente évaluation académique.

- **Robustesse et fiabilité** : Les invariants métier (contraintes sur les données, comme stock non négatif, etc.) sont centralisés dans le domaine et les use cases, ce qui réduit les risques d'erreurs. Par exemple, la classe `Product` empêche la création d'un produit invalide dès son constructeur. Les use cases, eux, empêcheront des opérations incohérentes (vente au-delà du stock disponible, etc.). Grâce aux tests possibles à chaque niveau, on peut attraper les bugs tôt. Globalement, l'architecture Clean fournit *« un cadre solide pour naviguer dans la complexité sans se perdre dans les détails »*, produisant un code à la fois **solide et souple** [28] [29].

Documentation Technique et Outils de Validation

Un aspect essentiel pour un rendu professionnel sera la **documentation** du code et de l'architecture. Après avoir implémenté cette conception, on réalisera une documentation technique complète afin de valoriser le travail effectué et de faciliter la maintenance future :

- **Documentation du code avec Doxygen** : Chaque classe, méthode et attribut sera commenté de manière descriptive (contrat, rôle, détails d'implémentation si nécessaires). À l'aide de Doxygen, on générera un document technique (probablement en HTML/PDF) présentant tous les éléments de l'API interne. Cette documentation servira de référence pour les utilisateurs du code ou pour les développeurs qui reprendront le projet. Elle attestera du sérieux apporté à la conception (en mettant en avant les descriptions de `Product`, `SimpleProduct`, `ContainerProduct`, des services de stock, etc., avec leurs invariants et préconditions).

- **Diagrammes UML** : En parallèle, on produira des diagrammes UML pour illustrer visuellement la structure et le comportement du système. Un **diagramme de classes** présentera l'organisation du domaine (`Product` ← `SimpleProduct`/`ContainerProduct`, association entre `ContainerProduct` et `Product` contenu, etc.), ainsi que les relations entre couches (par exemple, le service de stock reliant domaine et repository, l'interface UI dépendant du service, etc.). On veillera à bien indiquer la navigation des dépendances (flèches dirigées vers le centre pour respecter la Clean Architecture). De plus, un ou plusieurs **diagrammes de séquence** mettront en scène les interactions dynamiques pour des cas d'usage clés – par exemple, le scénario de vente d'un produit conteneur : l'UI console envoie la commande au contrôleur, qui invoque la méthode de vente du service applicatif, lequel va chercher le produit dans le repository, vérifier le stock, mettre à jour les entités (`ContainerProduct` et son produit contenu), puis persister les changements et renvoyer un résultat que l'UI affiche. Ces diagrammes permettront de vérifier que le flux d'information suit bien les couches prévues (UI -> use case -> domaine -> persistance) et de communiquer clairement le comportement du système aux examinateurs.

- **Relecture et validation croisée** : Enfin, avant rendu, il est prévu de passer en revue le code et l'architecture pour s'assurer que tout est cohérent avec les principes annoncés. Chaque couche sera vérifiée : par exemple, aucune classe du domaine ne doit dépendre d'une classe d'interface ou de persistance (respect de la règle de dépendance). Les interfaces seront correctement implémentées et injectées. Des cas de test en console seront exécutés pour démontrer concrètement le fonctionnement (ajout d'articles, ventes, etc., y compris les cas d'erreur à gérer comme tenter de vendre un stock insuffisant). Cette étape garantira que le résultat final n'est pas seulement théorique, mais aussi fonctionnellement abouti.

En conclusion, en combinant une **architecture logicielle exemplaire** et une **documentation professionnelle**, ce projet de gestion de stock se distinguera par sa qualité. La mise en œuvre de la Clean Architecture, alliée aux design patterns appropriés (tels que le composite pour les produits, le repository pour la persistance, etc.), offre un code soutenable, évolutif et facile à tester. C'est exactement le type d'approche qui mène à un logiciel robuste et bien noté. En suivant ce plan rigoureux, nous nous assurons de livrer un travail au **meilleur niveau**, conforme à l'exigence d'excellence annoncée. Les sources et principes étudiés (Clean Architecture [30] [14], principes SOLID, etc.) convergent tous vers un même but : produire un programme **propre** (« *Clean* »), tant dans sa structure que dans son exécution. Avec une telle architecture en place, nous sommes sur la bonne voie pour atteindre (et même dépasser) l'objectif de la meilleure note finale.

Sources Utilisées: Clean Architecture (principes et couches) [30] [13] [14] [7], notions de design pattern Composite [16], documentation et retours d'expérience sur la Clean Architecture [1] [4] [24]. (Voir les références intégrées pour plus de détails.)

---

[1] [2] [3] [4] [6] [8] [9] [10] [11] [12] [13] [15] [17] [18] [19] [20] [23] [24] [25] [26] [27] [28] [29] [30] Les fondements de la clean architecture: une voie vers un code durable | by Abderrahmane Roumane | Medium
https://medium.com/@abderrahmane.roumane.ext/les-fondements-de-la-clean-architecture-une-voie-vers-un-code-durable-1542f91f9d0a

[5] [7] [14] [21] [22] Quels sont les principes de la clean architecture ? - nava Design
https://www.navadesign.com/clean-architecture/

[16] Composite / Composite
https://refactoring.guru/fr/design-patterns/composite