Rayan ESSAIDI

SAE Lego : Composant Java

1. Introduction du projet

Ce projet vise à créer un tableau LEGO personnalisable à partir d'une image téléversée par un utilisateur.
Le composant Java agit comme intermédiaire entre le frontend PHP, le solveur de pavage C, la base de données, et l'API REST de l'usine de fabrication LEGO.

Objectifs principaux :

- **Flexibilité :** Permettre l'ajout simple de nouvelles fonctionnalités futures
- **Organisation :** Isoler les différentes fonctionnalités

2. Architecture du composant

Package	Rôle
brick	Contient les données essentielles aux pièces.
factory	Interface standardisée pour communiquer avec l'usine et gérer les paiements.
stock	Gestion du stock et des commandes si besoin.
image	Encapsule les différents algorithmes de redimensionnement.

A. Traitement d'Image :

- **Choix d'implémentation :** Les algorithmes NearestNeighborStrategy et BilinearStrategy sont implémentés manuellement sans l'utilisation de méthodes de scaling du package java.awt
- **Avantage de l'architecture :** La classe MultiPhasesStrategy utilise le Pattern Composite pour permettre à l'utilisateur de définir un pipeline de réduction (ex: Bilinéaire 1024->512 puis Nearest Neighbor 512->128), offrant une grande flexibilité d'expérimentation.

B. Intégration Usine :

- **Pattern Adapter (BrickFactory / FactoryConfig) :** L'interface BrickFactory définit le Port standard pour toute usine. FactoryConfig est l'Adapter qui implémente ce port en utilisant le protocole REST (via HttpURLConnection et Gson). Cela permet,

comme demandé, d'ajouter facilement un UdpFactoryAdapter si une autre usine utilise le protocole UDP.

- **Utilisation de Gson :** Gson est choisi pour sa simplicité et sa capacité à mapper facilement les objets Java aux structures JSON complexes requises par l'API de l'usine (QuoteRequestResponse, DeliveryResponse).

C. Gestion du Paiement :

Pattern Strategy (PaymentStrategy / PaymentService) : La stratégie de paiement est abstraite par l'interface PaymentStrategy. Actuellement, seule la stratégie PaymentService est implémentée.
Avantage de l'architecture : Si un jour un fournisseur de paiement par carte bancaire (via une autre API) était ajouté, il suffirait de créer une nouvelle classe CreditCardPaymentStrategy qui implémentent le même contrat rechargeAccount.

3. Choix d'implémentation spécifiques

Zone	Choix d'Implémentation	Justification
Briques	Utilisation des `record` Java	Les `records` sont parfaits pour représenter les modèles de données immuables comme les pièces.
API Usine	`HttpURLConnection` (JDK standard)	Respect de la contrainte initiale d'utiliser les outils standards du JDK, bien que des clients HTTP plus modernes existent. En revanche, cela nécessite une gestion manuelle des streams et codes de retour.
Constantes d'API	Classe `FactoryConfig` publique	Centralisation de l'e-mail, de la clé secrète et de l'URL de base pour une maintenance et un accès inter-classes simplifiés.
Gestion du Stock	`determineMissingBricks`	La méthode ne fait pas que déterminer le manque ; elle réserve aussi immédiatement le stock disponible pour éviter la survente en cas de commandes concurrentes.
billing/challenge	`java.security.MessageDigest`	Utilisation du package standard du JDK pour le hachage SHA-256, garantissant la portabilité et la sécurité du calcul cryptographique.

4. Difficultés rencontrés et défis potentiels

1. **Gestion de la concurrence (Stock) :** L'implémentation actuelle de determineMissingBricks met à jour le stock en soustrayant les pièces utilisées. Dans un environnement multithread (avec plusieurs clients qui commandent en même temps), une simple soustraction n'est pas suffisante. Il faudra implémenter un mécanisme de verrouillage pessimiste (via la base de données) ou s'assurer que l'opération updateStock est atomique et transactionnelle pour éviter les cas de concurrence.
2. **Gestion des erreurs API :** FactoryConfig ne gère actuellement que les codes de succès ou les échecs simples (200, 402, 404…). Une gestion plus robuste des erreurs, incluant la désérialisation des corps de réponse d'erreur et une logique de retry (tentative de nouvelle connexion) pour le code 500, sera nécessaire.
3. **Interface C/Java :** La communication avec le solveur C est une étape critique à implémenter. Ce pont technique peut introduire des défis de performance, de sérialisation des données, et de gestion des processus externes.
4. **Manque de pratique :** Principalement sur la gestion des stocks, le travail à effectuer nécessite des connaissances que récemment ou pas encore vus en cours, et pas encore pratiqués ni acquis.
5. **Manque de temps :** De pair avec la difficulté précédente, avec l'avancé des cours magistraux, le développement sur ce module a commencé assez tard par rapport à la date de rendu, ce qui rend le travail d'implémenter toutes les fonctionnalités demandés très difficile. (P.S : je profite pour m'excuser du retard du rendu dû au temps consacré à écrire la documentation après avoir sécurisé le rendu des autres modules)

5. Prochaines étapes définies (À implémenter)

- **Quantification des couleurs et des formes :** Mappage des pixels de l'image réduite à la palette de couleurs et aux formes de pièces Lego disponibles en stock.
- **Intégration C :** Mise en place d'une fonctionnalité pour communiquer avec le programme de pavage.
- **Génération du Tableau :** Création de l'image de prévisualisation finale du tableau basé sur le résultat du solveur C.
- **Exemples avec des main :** Ajout de méthodes main pour donner des exemples d'utilisation en exécutant directement le programme Java. (Par manque de temps, le développement des fonctionnalités avec de petits tests a été priorisé sans laisser le temps à une réflexion d'un main satisfaisant)