TP : FoodFast

Bienvenue dans l'équipe de développement de **FoodFast**, une startup innovante qui révolutionne la livraison de repas. Suite à une croissance rapide, FoodFast doit améliorer son système de gestion des commandes et des livraisons pour offrir un service plus performant aux restaurants partenaires et aux clients finaux.

L'équipe technique est mobilisée pour concevoir une solution à la hauteur de ces nouveaux défis. Vous faites partie de cette équipe et votre mission est de créer un prototype qui pourra ensuite être intégré au système de production.

Objectif du Projet

FoodFast reçoit de plus en plus de commandes et doit s'assurer que chaque commande est livrée dans les meilleures conditions. Vous allez développer un prototype pour simuler l'envoi et la gestion des commandes entre plusieurs restaurants et une plateforme de livraison centralisée. Votre solution devra évoluer au fur et à mesure des exigences pour être plus robuste et flexible.

Votre première tâche est de concevoir une solution simple. Un restaurant doit pouvoir générer une commande en fonction d'un délai d'attente aléatoire et l'envoyer à une plateforme de livraison. La plateforme de livraison reçoit les commandes et les gère pour les dispatcher.

Question 1 : Créer les classes **Application.java**, **Restaurant.java**, **DeliveryPlateform.java, Order.java** et **Dish.java**

La classe Dish.java doit contenir un nom et d'une taille (enum S,M et L)

La classe Order.java doit contenir le restaurant de préparation, des plats commandés (Dish.java) et de leurs quantités, d'un montant initial et d'un lieu de livraison (dans ce cas un String est suffisant).

La classe Restaurant, doit contenir un nom et une méthode prepareOrder qui génère et retourne l'order (vous pouvez ajouter un appel à la méthode Thread.sleep(new Random().nexInt(3000)) pour simuler un temps d'attente.

La classe DeliveryPlateform qui contient une méthode "void delivery(Order order)" simulant la livraison d'une commande.

La classe Application.java, qui dans la méthode main, instancie une plateforme de livraison, un restaurant, et effectue quelques commandes.

Écrivez les tests unitaires associées aux méthodes.

Question 2 :

Après avoir observé la première version, l'architecte de FoodFast remarque que la solution manque de réactivité et propose d'introduire un modèle Observer. Ce modèle permettra à la plateforme d'être informée en temps réel des commandes générées par les restaurants, sans que le restaurant ait besoin de connaître les détails internes de la plateforme.
Classe à modifier :

1. **Restaurant** : notifier la plateforme dès qu'une commande est générée.

Classe à modifier :

1. **DeliveryPlatform** : s'abonner pour recevoir les notifications des commandes en temps réel.

Implémentez le pattern Observer afin que le restaurant notifie la plateforme de livraison (https://en.wikipedia.org/wiki/Observer_pattern)

Écrivez les tests unitaires associées aux méthodes.

Question 3 :

FoodFast collabore avec plusieurs restaurants, et chaque restaurant doit pouvoir s'abonner à la plateforme de livraison. Votre prototype doit donc maintenant gérer les commandes provenant de plusieurs sources de manière concurrente.
Classe à modifier :

1. **DeliveryPlatform** : gérer plusieurs restaurants, chacun pouvant s'abonner à la plateforme.

Question 4 :

Avec l'augmentation du volume de commandes, l'équipe technique remarque des doublons dans les commandes, ce qui cause des erreurs. Il est donc crucial de garantir que chaque commande soit unique pour éviter tout bug.
Classe à modifier :

1. **DeliveryPlatform** : assurer que chaque commande reçue soit unique.

Comment pourriez-vous gérer l'unicité des commandes et éviter les doublons ?

Écrivez les tests unitaires associées aux méthodes.

Question 5 :
FoodFast souhaite maintenant que les commandes soient traitées dans un ordre bien défini pour garantir une meilleure expérience client. L'ordonnancement des commandes est crucial pour leur bonne gestion.
Comment organiseriez-vous l'ordonnancement des commandes pour qu'elles soient traitées dans l'ordre de priorité ? (Indice: lisez la documentation de la classe PriorityQueue)

Question 6 :

Avec la croissance exponentielle de FoodFast, le système doit devenir plus évolutif. Il est désormais nécessaire de passer à une architecture plus souple, capable de supporter des dizaines de restaurants et des centaines de commandes en même temps. L'architecte propose d'introduire un **EventBus** pour gérer cette complexité.
Comment concevriez-vous un système capable de gérer l'augmentation massive des commandes via un EventBus ?

Question 7 :

a) Dans un environnement de production réel, des erreurs peuvent survenir, telles qu'un restaurant qui échoue à envoyer une commande ou une plateforme de livraison qui ne traite pas correctement une commande. Il est essentiel de gérer ces erreurs pour garantir la stabilité du système.

Créer les erreurs liées à la préparation et à la livraison de la commande et envoyées les de façons aléatoires dans le code et implémenter la gestion d'erreur.

b) (Question bonus) Afin d'améliorer l'observabilité du système, l'eventbus doit garder en mémoire l'intégralité des messages reçus et pouvoir les rejouer une seconde fois en cas d'erreur.

Question 8 :

Comment pouvez-vous concevoir une interface Event en Java pour représenter une abstraction des différents types d'événements que l'EventBus doit gérer, tels que des commandes, des retours et l'envoi de factures entre fournisseurs ? (OrderEvent, ReturnEvent, InvoiceEvent)

Question 9 :

Refactorer l'intégralité du code en utilisant exclusivement les stream, optional.

Question 10 :

Créer un nouveau service de facturation, qui à la réception d'un nouvel event (OrderDelivered) créée une facture, gardée en mémoire et envoyée (ici nous nous contenterons d'un log "Invoice sent".