Conseil : À l'examen commence par lire d'abord le fichier des tests, il te permettrons de mieux comprendre l'exo. Ou bien essayer de représenter les classes en des tables pour mieux comprendre le lien entre eux UML

Quand tu fais le refactoring, guide-toi et pour chaque amélioration majeure fais un commit

Après les grandes étapes et l'application du design pattern

On peut penser a Ensuite séparer les test en plusieurs test et vérifier a chaque fois, faire un commit

Puis créer les tests pour chaque classe

--

TPs-revision

L'objectif est de pratiquer des techniques de refactoring simples pour améliorer la qualité du code tout en conservant son fonctionnement.

Étape 1 : Vérifier les Tests Existant

Avant de commencer à refactoriser, exécute tous les tests unitaires pour t'assurer qu'ils passent tous.

->Si possible même refactorer le test en testant chaque partie par exemple :

Tester Séparément le Montant dû pour Chaque Type de Film

git commit -m « refactoring pour Décomposer les Tests »

->Sinon suivre les autres étapes et attendre à la fin

Étape 1.1 (phase d'examination 2 à 4 minutes) : examiner la structure et l'organisation générale du code

Point de départ. Avant d'examiner le code interne des classes, il est essentiel de s'assurer que l'architecture globale du système est bien conçue

Examiner l'Architecture du Projet :

Structure des Packages : Vérifie si les classes sont regroupées logiquement dans des packages qui reflètent leur fonction dans l'application. (On ne considererai pas cela au controle)

Point d'Entrée : Cherche le point d'entrée de l'application (habituellement une méthode main dans une classe Main ou Application). S'il n'existe pas, envisage de le créer pour démarrer l'application et orchestrer les interactions entre les objets.

Cohésion et Couplage : Assure-toi que les classes sont fortement cohésives (c'est-à-dire, elles ont une seule responsabilité) et faiblement couplées (c'est-à-dire, elles ne dépendent pas trop des détails des autres classes). (On va régler cela en appliquand le design pattern adéquat)

Examiner les Principes de Conception :

Principes SOLID : Évalue si les principes SOLID sont respectés. Par exemple, chaque classe devrait avoir une seule raison de changer (Single Responsibility Principle), et tu devrais pouvoir étendre le comportement d'une classe sans la modifier (Open/Closed Principle).

Principes DRY et KISS : Cherche des duplications de code qui pourraient être éliminées (DRY) et vérifie si le code est simple et direct (KISS).

Patterns de Conception : Identifie si des design patterns peuvent être appliqués pour améliorer la structure ou le comportement du code, comme le pattern Strategy pour des opérations polymorphiques ou le pattern Factory pour la création d'objets.

Étape 2 : Analyser le Code

Passe en revue le code source de chaque classe (Customer, Movie, Rental) et identifie les "code smells" (opportunités de refactoring).

Regarde pour les méthodes longues, les méthodes avec de nombreux paramètres, les variables mal nommées, les duplications de code, etc

Analyser la conception de l'ensemble de code, quel est le modele de conception est ce que les classe sont bien organiser, est ce qu'il y a un main.

→ *Plusieurs étapes avant d'arrivée à l'étape 3 et ces étapes sont détaillé après la barre en bas*

Étape 3 : Refactoriser

Commence par des refactorisations simples comme renommer les variables pour plus de clarté, extraire des constantes, ou diviser les méthodes complexes.

Après chaque modification, re-exécute les tests pour t'assurer que tu n'as pas introduit d'erreurs.

Étape 4 : Ajouter la Fonctionnalité

Une fois le refactoring terminé, implémente la nouvelle fonctionnalité demandée éventuellement.

Assure-toi que cette nouvelle fonctionnalité est également bien testée.

Étape 5 : Documentation et Commit

Documente chaque étape de refactoring que tu fais avec des commits dans Git.

Utilise des messages de commit clairs qui expliquent quel refactoring tu as effectué.

Les étapes seront toujours de cette façons et suivrons cette logique :

Étape 1: Créer la Classe Main

Crée un fichier Main.java dans le répertoire src/main/java.

Dans Main.java, écris une méthode public static void main(String[] args) qui servira de point d'entrée pour ton application.

Implémente la logique nécessaire pour initialiser les objets et démarrer le processus de location des films.

Étape 2: Refactoring de la Classe Customer

Sépare la méthode statement() en plusieurs méthodes plus petites pour respecter le principe de responsabilité unique.

Crée des méthodes pour calculer le montant de chaque Rental, les points de locataire fréquent et construire le résultat final.

Étape 3: Application du design Pattern (Introduction du Polymorphisme pour les Prix des Films)

Refactore les types de prix dans Movie pour utiliser le polymorphisme au lieu des constantes et de l'instruction switch.

Crée une classe ou une interface Price et des sous-classes spécifiques pour chaque type de film (Regular, New Release, Children's).

Intègre cette nouvelle structure dans la méthode de calcul des montants dans Customer.

Étape 4: Refactoring de la Classe Movie

Élimine la méthode setPriceCode(int arg) si le prix du film ne change pas après sa création.

Assure-toi que la classe Movie utilise la nouvelle structure de Price pour le calcul du prix.

Étape 5: Améliorer la Classe Rental

Pense à déplacer une partie de la logique de calcul des frais de Customer vers Rental si cela améliore la cohésion.

Étape 6: Créer et Exécuter des Tests pour la Classe Movie et Rental

Écris des tests unitaires pour Movie et Rental qui couvrent toutes les nouvelles logiques que tu as introduites.

Étape 7: Exécuter Tous les Tests et Vérifier la Couverture

Une fois tous les refactoring effectués, exécute tous les tests pour s'assurer qu'il n'y a pas de régression.

Vérifie la couverture de test pour t'assurer que toutes les parties du code sont bien testées.

Étape 8: Révision et Nettoyage ->qui correspondait à l'Étape 3 : Refactoriser

Passe en revue tout le code pour le nettoyage final. Assure-toi que tous les commentaires, noms de variables et méthodes sont clairs et précis.

Vérifie que le code est bien organisé et que les principes de conception sont correctement appliqués.

IntelliJ propose un plugin Checkstyle qui peut être installé via le gestionnaire de plugins. Après l'installation, tu peux configurer le plugin dans Settings > Tools > Checkstyle

Cherche l'option qui active Checkstyle pour s'exécuter sur le code à chaque compilation ou lors d'autres événements spécifiques, et assure-toi qu'elle est activée.

Étape 9: Documentation

Documente tous les changements importants dans ton code, y compris pourquoi et comment tu as fait ces changements pour que les autres développeurs ou toi-même dans le futur puissent comprendre les raisons derrière ces refactoring.

--