Info – Bloc 1 – UE09 : Programmation Orientée Objet

Laboratoire 5

Durée prévue : 4 heures

Objectifs visés

À la fin du laboratoire, les étudiants seront capables d'utiliser PMD pour améliorer la qualité de leur code. Ils seront également aptes à manipuler des collections en exploitant le polymorphisme par sous-typage, sans nécessairement chercher à choisir la plus adaptée.

Retour sur le labo 4

Durée estimée : 10 minutes

Le responsable de laboratoire présente une correction pour le labo 4. Il insiste notamment sur les mécanismes de polymorphisme par sous-typage, qui font qu'un appel à la méthode useForceOn(BaseCharacter) depuis une référence à un BaseCharacter sera traité différemment selon la classe concrète de l'objet qui recevra cet appel.

Créer le projet

Durée estimée : 05 minutes

Crée un projet Eclipse intitulé `poo.labo05`. Dans les propriétés du projet :

- Ajoute un répertoire de sources pour les tests.
- Active PMD et renseigne le fichier de règles *poo-labo05-ruleset.xml*, disponible en annexe.
- Ajoute JUnit 5 comme dépendance au projet

Ajoute enfin les fichiers proposés en annexe. Tu devrais obtenir la structure de départ suivante.

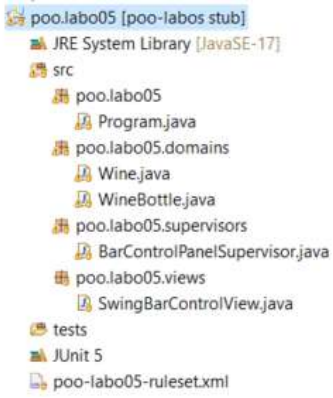


Figure 1 Structure attendue

Exercices

La gérante d'un bar à vins désire une application gérant les ventes d'une journée. Tous les matins, la gérante remet son stock à jour. Son bar à vin autorise la vente en bouteille et au verre tant que le stock le permet.

L'application désirée doit afficher l'état du stock et le tenir à jour en fonction des consommations encodées.

Un jeune analyste-programmeur a mis au point une première version du programme. Cette dernière fonctionne, mais son code présente une structure discutable. Pour résoudre ces problèmes, vous utiliserez PMD et ses alertes.

Exercice 1 : concevoir une classe Stock

Durée estimée : 20 minutes

Objectifs :

- Identifier les responsabilités
- Identifier les morceaux de code associés aux responsabilités

Actuellement, la classe `poo.labo05.supervisors.BarControlPanelSupervisor` est responsable d'alimenter l'interface utilisateur en données à afficher, de réagir aux demandes de l'utilisateur et de maintenir l'état du stock : *« bien trop de responsabilités pour un seul homme »*. Pour la soulager, nous souhaitons déplacer la logique de gestion du stock dans une classe dédiée.

Nous voulons une classe représentant un stock capable de :

- Fournir les vins afin de les parcourir
- Pour chaque vin, fournir le nombre de bouteilles pleines, le prix et le volume restant
- Diminuer le stock de X bouteilles
- Diminuer le stock de X verres

Le superviseur sera toujours responsable de créer les tables d'affichages et la liste des libellés de vin et de les fournir à sa vue. Il devra en outre contrôler l'objet stock pour répondre aux demandes de l'utilisateur.

Commence par identifier les futures responsabilités de chaque classe en complétant le tableau suivant. Identifie ensuite, pour chaque responsabilité, les lignes de code correspondantes dans la classe `BarControlPanelSupervisor` existante.

`BarControlPanelSupervisor`		`Stock`	
Fournir la table représentant le stock		Fournir les vins	

Retour avec le responsable

Durée estimée : 20 minutes

Le responsable présente une correction. À partir de cette correction, il déclinera les responsabilités en méthodes à implémenter.

Exercice 2 : implémentations

Durée estimée : 60 minutes

Objectifs :

- Extraire des méthodes
- Encapsuler les attributs et utiliser des copies défensives
- Masquer les détails d'implémentation

Définis la classe `poo.labo05.domains.Stock`. Cette classe aura un constructeur prenant une **`Collection de bouteilles`** en paramètre. Elle exposera au superviseur les méthodes nécessaires et suffisantes à l'affichage des données et aux réactions aux demandes de l'utilisateur.

Veille à ce que ta classe masque l'utilisation de la map et des listes de bouteille. Assure-toi aussi de ne pas retourner des références aux collections de l'objet. En particulier, retourner directement le résultat de `keySet()` est-il sûr ? Pour répondre à cette question, consulte la Javadoc de Map.keySet().

Écris des tests unitaires pour valider la classe Stock. Intéresse-toi particulièrement aux cas où le stock pour une bouteille donnée est vide ou insuffisant.

Quand ta classe te semble au point, instancie-la dans la classe `poo.labo05.Program` et transmets-la au superviseur.

À la fin de ton remaniement, PMD ne devrait afficher aucune alerte ni pour `BarControlSupervisor`, ni pour `Stock`.

Si l'horaire le permet, le remaniement est à terminer pour la séance suivante.

Retour avec le responsable

Durée estimée : 15 minutes

Le responsable présente une correction pour la classe Stock. Il insiste sur le fait que cette classe est responsable de son état et qu'elle ne dépend pas de l'interface utilisateur, ce qui la rend réutilisable.

Exercice 3 : Remanier la classe des vins

Durée estimée : 45 minutes

Objectifs :

- Définir une classe d'objets valeurs
- Appliquer le principe *Tell don't ask* en s'aidant de *code smell*

Les objets `Wine` ne changent pas d'état après leur création. Elle est une bonne candidate à une transformation en classe d'objets valeurs, qui sont immuables. Commence par transformer la classe des vins en classe d'objet valeurs.

Astuce

Tu ne sais pas ce qu'est une classe d'objet valeurs ? Consulte cette page pour en apprendre plus sur leurs caractéristiques.

Malgré tes transformations, `Wine` reste une classe de données. En parallèle, certaines instructions de `Stock` jalousent les données de `Wine`. Résous ces problèmes en ajoutant de nouvelles méthodes à la classe `Wine` sur base de ces indicateurs.

Astuce

Tu ignores ce que sont les *odeurs de code ("code smell")* *data class* et *feature envy* ? Consulte les pages suivantes :

- Data class: https://refactoring.guru/fr/smells/data-class
- Feature envy: https://refactoring.guru/fr/smells/feature-envy

Écris quelques tests unitaires pour valider le comportement de ta classe. À ce stade, tu ne devrais plus avoir d'alertes PMD pour `Wine`. Tu peux te passer des accesseurs.

Exercice 4 : Remanier la classe des bouteilles de vin

Durée estimée : 30 minutes

Objectif : encapsuler les objets

Contrairement aux objets de la classe `Wine`, les bouteilles de vin changent d'état : on peut vider leur contenu. Actuellement, `Stock` manipule directement le volume des bouteilles, ce qui peut poser des problèmes de cohérence des données.

Encapsule la classe `WineBottle` en inspectant l'utilisation qu'en fait la classe `Stock`. À quelles questions correspondent les conditions qui portent sur le volume d'une bouteille ? Crée les accesseurs correspondants.

Quid des instructions qui modifient le volume restant d'une bouteille ? Que représentent-elles ? Crée des méthodes correspondantes.

Après tes modifications, aucune alerte PMD ne portera sur `WineBottle`. As-tu écrit quelques tests unitaires pour valider tes changements ?

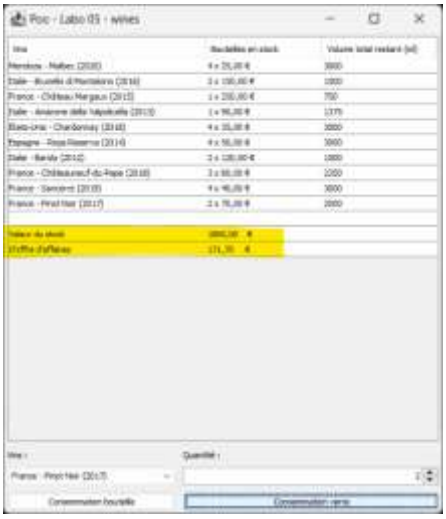
Exercice 5 : calculer les consommations

Durée estimée : 30 minutes

Objectifs :

- Définir une hiérarchie d'héritage
- Appliquer le principe *Tell don't ask* et *Loi de Déméter*

Jusqu'à présent, vous avez remanié un code existant. Toutefois, la gérante souhaite connaître la valeur totale du stock et son chiffre d'affaires après chaque consommation. Ces valeurs seront ajoutées en fin de table.



La valeur du stock correspond à la somme des prix d'achat des bouteilles pleines[1]. Le chiffre d'affaires correspond à la somme des prix de vente des consommations. Enfin, le prix de vente dépend du type de consommation (au verre ou à la bouteille).

Le bar vend toute bouteille à 120 % de son prix d'achat. En revanche, le prix de vente au verre dépend du prix d'achat de la bouteille, rapporté au volume d'un verre (125 ml pour un verre, et 750 ml pour une bouteille). Plus une bouteille coute cher, plus le prix de vente au verre augmente. La table suivante présente les marges.

Prix d'achat	Pondération pour le prix au verre
< 50 €	156 %
< 100 €	166 %
< 1000 €	199 %

Si une bouteille est achetée 300 €, son prix de vente à la bouteille sera de 360 €. Son prix au verre sera de 300 * (125/750) * 199 % = 99,5 €. Une bouteille achetée 35 € sera vendue 42 €. Son prix au verre sera de 35 € * (125/750) * 156 % = 9,1 €.

Remarque

Le bar veut afficher des prix au dixième d'euro. Si un prix compte des centièmes d'euros, il sera arrondi au dixième supérieur. Ainsi, 9,01 € et 9,09 € seront arrondis à 9,1 €. Toutefois, 9,009 € sera arrondi à 9,00 €.

A priori, deux méthodes suffisent à calculer les prix de vente : une pour consommer la bouteille et une pour les consommations au verre. Ces méthodes représentent de la logique métier, les règles sont spécifiques au bar et indépendantes de l'interface utilisateur. Déclare-les dans une classe poo.labo05.domains.Consumption.

Avant de commencer à coder, réponds aux questions suivantes ?

[1] Les bouteilles ouvertes, mais pas vides, sont utilisées pour faire du vinaigre.

- Quelle classe connait toutes les données nécessaires et suffisantes pour calculer la valeur du stock ?
- De quels paramètres les méthodes de calcul des consommations ont-elles besoin ? Serait-il nécessaire de transmettre une bouteille entière ? Veille à réduire le couplage au maximum.
- Dois-tu modifier la classe `Stock` ? Si oui, tâche de continuer à respecter les principes *Tell don't ask* et *Loi de Déméter*.

Pense à tester tes classes !

Pour les plus rapides, une seconde manière de modéliser les stratégies de calcul des consommations serait d'utiliser une énumération déclarant une méthode abstraite. Inspire-toi de la page donnée en lien ci-dessous pour essayer d'implémenter cette approche.

https://riptutorial.com/java/example/3276/enums-with-abstract-methods