Cours 420-266-LI Programmation orientée objet II Hiver 2024 Cégep Limoilou Département d'informatique Professeur : Martin Simoneau	TP1 (10 %) Héritage - interface Polymorphisme Abstractions

Objectifs

- Réutiliser et rendre compatible en utilisant :
  - Héritage
  - Interface
  - Délégation
  - Abstraction
  - Polymorphisme

Consignes :

- Remettre vos projets sur Omnivox/*Léa* à la date indiquée.
- Le travail se fait seul.
- La copie est interdite. Tout code provenant d'une aide extérieure (ex.: ami, famille, *StackOverflow*, IA…) qui n'est pas mentionnée directement dans le code est considéré comme de la triche et occasionnera la note 0 pour le TP1. Vous n'aurez aucun point pour les parties faites par une aide extérieure.
- Les tuteurs du cégep n'ont pas le droit de vous assister pour faire les TPs.
- Toujours ouvrir le dossier qui contient le fichier pom.xml.
- **Surveillez le canal TP1 sur Teams pour des mises à jour ou des modifications de l'énoncé.**

Contexte :

On reprend le garage que nous avons fait dans l'exercice de révision et on le combine avec le formatif 3 (Camion, bolide et Automobile). Nous allons donc faire rouler les 3 types de véhicules. Les véhicules sont endommagés lorsqu'ils roulent trop longtemps. On les envoie ensuite au garage pour être réparés.

À faire

- Garder une copie de sécurité du projet à titre de référence
- Vous pouvez modifier le code source, mais vous devez garder les fonctionnalités initiales intact (sauf si des changements sont demandés). Dans le doute, faites valider votre idée par le professeur.
- Vous avez le droit de vous créer d'autres package au besoin.

- Code fourni à modifier:
  - Dans le package tp1, vous trouverez la classe ***Application*** qui contient le scénario principal dans la méthode *main*. Au départ, le scénario ne fonctionne qu'avec les Automobiles. Vous pourrez l'adapter pour fonctionner avec les 3 types de véhicules au moment approprié.
  - Ajustement du code pour Automobile, *Bolide* et *Camion*
    - Bolide :
      - Méthode void ***roule***(double) : Le Bolide possède un attribut *estEnCourse* qui indique si la voiture circule normalement ou si elle est en course. Lorsque la voiture est en course, son kilométrage doit être doublé.
      - Méthode ***double repare***() : Lorsqu'un bolide roule plus de 150 kilomètres sans être réparé, le coût des réparations devient quadratique (coût exposant 2 ou coût ^{2}).
    - Automobile :
      - Pas de changement pour la méthode ***roule(double)***.

- Méthode double ***double repare()*** : Les automobiles simples ont une garantie qui couvre un montant de base. Tant que le montant de base n'est pas épuisé, le coût de réparation demeure 0. Évidemment, après quelques réparations, le montant de base sera épuisé et il faudra commencer à retourner un coût. Vous aurez certainement à ajouter un ou plusieurs attributs pour gérer cette garantie.

- Camion : Le camion est différent de *l'Automobile* et du *Bolide* parce qu'il dépend de **l'usure** plutôt que le kilométrage pour la gestion des bris et des réparations. L'usure diffère selon le type de chargement du camion.
  - 2 types de chargement de *Camion* :
    - Les 2 types de chargement dépendent de leur volume. Il faut donc retenir le volume de chaque chargement.
    - Avec une ***boîte*** directement sur le camion (type Boite), l'usure sera donnée par :
      - $usure = \frac{kilometrage * pourcentage\ occupé * volume\ total}{100}$
    - Avec une ***remorque*** l'usure sera donnée par
      - $usure = \max_{5} kilometrage * ratio\ appuie * volume\ total$

        Maximum entre 5 et le résultat du calcul décrit.
    - Gérer bien les principes **SOL**ID avec les types de chargement !
  - Méthode ***roule(double)*** :

    N.B. Même si l'on ne fait pas les calculs directement avec le *kilométrage*, le camion accumule quand même son *kilométrage*. Il sera utile pour le rapport d'assurance un peu plus loin.
    - À chaque 10km, on doit ajouter un point d'usure.
    - En plus, il faut ajouter l'usure causée par le chargement du camion.
    - Le camion est considéré comme brisé lorsque l'usure dépasse 50 et très brisé lorsque l'usure dépasse 125.
  - Méthode ***double repare()*** :
    - Le camion est réparé comme l'auto, mais son coût de base est le maximum entre ***5 fois le coût de base*** et ***l'usure fois 2***. (déjà fait)

  - Vous trouverez les 3 types de *véhicules* (*Automobile*, *Camion* et *Bolide*) dans le package *vehicule*. Vous constaterez qu'on n'a fait aucune réutilisation et que les méthodes ne sont pas compatibles. Vous devez faire le nécessaire pour réutiliser le code au maximum et pour rendre les méthodes *roule* et repare compatibles en respectant les principes OO **SOL**ID.
  - Adaptez le code du garage pour qu'il puisse recevoir et réparer tous les types de véhicules.

- La garantie de *l'automobile* est gérée par le garage. Au contraire, les garanties du *camion* et du *bolide* sont gérées par le propriétaire du véhicule. Pour aider les propriétaires avec leur réclamation. Les classes *Bolide* et *Camion* peuvent générer un rapport avec la méthode *genereRapport*(). Il serait bon de rendre cette méthode compatible et de réutiliser le code qui génère le rapport. Le rapport retourne la ligne suivante :
  - `Le garage a chargé «prix» en frais de réparation.`
  - Ici `«prix»` est le vrai coût total pour le véhicule.

- Scénario :

  Le scénario se répète 10 fois. À chaque nouvelle itération, la distance prévue augmente de 20km. La **distance prévue** est une variable utilisée dans le scénario.
  - Vous devez compléter le scénario fourni dans la classe *Application* :
    - Tous les véhicules doivent rouler la **distance prévue**.

        - Les bolides font une course d'une longueur égale à la **distance prévue**.
        - On place tous les véhicules dans le stationnement.
        - On entre les véhicules 2 par 2 dans le garage.
        - On les répare.
        - On les replace dans le stationnement.
        - On fait leur départ.
    - Valider que votre sortie en console correspond bon bien à celle qu'on vous a remise.

Critères d'évaluation et exigences:

Exigences

- Mettre vos noms en commentaires dans l'en-tête de chacune des classes dans lequel vous avez travaillé.
- Respectez la date de remise.

Qualité du code

- **Commentaires** pertinents et suffisants
- Les **noms** des méthodes, des classes et des attributs sont pertinents et **complets**.
- Toutes les méthodes sont protégées par les **assertions** adéquates.
- Il n'y a **pas** de **code inutile** ou commenté.
- **Formatage** effectué dans chaque classe.
- **Algorithmes simples** et efficaces.
- Le code fonctionne **sans erreurs**.
- **Toutes** les tâches demandées ont été accomplies.

- Classification et polymorphisme
    - La réutilisation et la compatibilité sont sans failles. Respect des 3 premiers principes OO **SOL**ID (Single-responsibility, open-closed, Liskov substitution principle)
    - L'annotation ***@Override*** est toujours utilisée lorsque c'est possible
    - **L'abstraction** est toujours utilisée lorsqu'elle est pertinente (abstract).
    - Le **code original** est **respecté** (pas de modification inutile du code original, pas de changement de fonctionnalité) sauf lorsque c'est explicitement demandé.

FIN