IUT Montpellier-Sète
Département Informatique
M. Bougeret, S. Gagné, S. Nabitz, J. Palaysi, V. Poupet, P. Valicov

TP 5 : Gestion d'employés

Héritage : réutiliser, mais sans en abuser

Avant de démarrer le TP, vérifiez que vous n'avez pas atteint votre quota d'espace de stockage autorisé et libérez l'espace si nécessaire.

CONSIGNES :

- Vous respecterez toutes les consignes indiquées dans le TP précédent
- Dans ce TP, un principe important que vous devriez essayer de respecter dans votre code est le principe DRY

Le site du TP où vous allez trouver le lien pour forker :

`https://github.com/IUTInfoMontp-M2103/TP5`

Date limite de rendu de votre code sur le dépôt GitHub : **Dimanche *** à 23h00**

Vous êtes chargés de proposer une application de gestion des employés dans une entreprise. L'objectif est de développer votre application de manière incrémentale, en ajoutant les fonctionnalités demandées au fur et à mesure **sans modifier les fonctionnalités écrites précédemment**. Ce que l'on considère ici comme *modification* c'est effacer et/ou réécrire du code précédemment écrit. *Ajouter* du code sans modifier le code précédent est donc une opération valide. On dira ici, que pour chaque modification de votre programme (effacement et réécriture) la dette de votre logiciel augmente. Afin de respecter les divers principes en programmation orientée objets (encapsulation, DRY, YAGNI etc.), vous essayerez d'éviter au maximum de *modifier* le programme écrit précédemment pour ne pas trop augmenter cette dette...

Exercice 1

1. Dans un premier temps vous devez modéliser les employés qui sont représentés par les données suivantes : numéro de sécurité sociale, nom, prénom, échelon (entier naturel). Implémentez la classe `Employe` correspondante et ajoutez-y un constructeur approprié. Le constructeur ayant beaucoup de paramètres, il vous est également demandé de proposer un *builder* pour une construction plus souple (cf. question 5, Exercice 3 du TP3). Vous pouvez générer le builder de manière automatique avec l'outil de refactoring d'IntelliJ IDEA :
   — clic droit sur le nom du constructeur de votre classe $\longrightarrow$ *Refactor* $\longrightarrow$ *Replace Constructor with Builder*
   — Dans la fenêtre qui s'affiche vous cocherez tous les paramètres comme indiqué dans le dessin ci-dessous et cliquerez sur le bouton *Refactor*
2. On souhaite pouvoir calculer le salaire brut et le salaire net d'un employé. Le salaire brut de l'employé se calcule de la manière suivante : `base * nbHeures`, où base et `nbHeures` seront des attributs de type double. Le salaire net représentera toujours 80% du salaire brut. Ajoutez le code nécessaire pour intégrer ces fonctionnalités.
3. Redéfinissez la méthode `toString()` dans la classe `Employe` pour afficher les informations concernant un employé (y compris ses salaires brut et net).
4. Si votre client vous demande de changer (modifier donc) la formule de calcul du salaire brut et la fixer à `base * nbHeures * 1.05`, combien de modifications devez-vous faire pour que votre programme continue de fonctionner correctement ? Est-ce que vous pouvez faire mieux ? **Remarque** : dans ce qui suit, le salaire brut d'un employé restera toujours le même, à savoir `base * nbHeures`.
5. Vérifiez votre solution dans le programme principal (la classe `GestionEmployes`). Vous y instancierez plusieurs employés (avec le builder) et afficherez les informations les concernant.

Exercice 2

1. Maintenant votre client se rend compte qu'une séparation des traitements est nécessaire pour les différents types d'employés. Il faut spécifier les cas des *Commerciaux*, *Fabricants*, et les autres employés qu'on appellera *Techniciens*.
   — Un `Commercial` a comme attributs `chiffreAffaires` et `tauxCommission` (tous les deux de type double).
   — Un `Fabricant` a comme attributs `nbUnitesProduites` et `tauxCommissionUnite` (type `int` et `double` respectivement).
   — Un `Technicien` n'a pour l'instant aucun nouveau attribut, ni aucune nouvelle méthode. Implémentez les classes correspondantes en les faisant hériter de la classe `Employe`.
2. Un commercial peut négocier des transactions (avec la méthode `negocierTransaction()`), un fabricant fabrique des produits (méthode `fabriquerProduit()`), un technicien effectue les autres tâches dans l'entreprise (méthode `effectuerTacheTechnique()`). Chacune de ces méthodes sont de type void et se contentent d'afficher un message approprié pour illustrer leur bon fonctionnement. Par exemple, la méthode `negocierTransaction()` devra afficher "Je négocie une transaction".
3. Vérifiez votre programme dans la classe principale, en instanciant un objet pour chaque nouveau type d'employé et en appelant sa fonction spécifique.
4. On souhaite varier le calcul des salaires bruts des différents types d'employés :
   — Le salaire brut d'un technicien est composé de son salaire brut en tant qu'employé + une majoration en fonction de son échelon. Dans notre exemple, le résultat de ce calcul devrait correspondre à `base * nbHeures + echelon * 100`.
   — Le salaire brut d'un commercial dépend du chiffre d'affaires qu'il réalise. Ainsi, le salaire brut se calcule suivant la formule `base + chiffreAffaires * tauxCommission`.
   — Le salaire brut d'un fabricant est calculé de la même manière que le salaire brut d'un employé en ajoutant une rémunération supplémentaire en fonction du rendement. Dans notre exemple, le résultat de ce calcul devrait correspondre à `base * nbHeures + nbUnitesProduites * tauxCommissionUnite`.
   — **Important** : La modalité de calcul du salaire net demeure inchangée pour tous les employés (à savoir 80% du salaire brut).

   Redéfinissez la méthode calcul de salaire brut dans chaque classe d'employé spécifique pour prendre en compte ces nouvelles formules. Vous ajouterez le code qui vous paraît nécessaire à la classe `Employe` mais sans modifier le code précédemment écrit.
5. Le patron est devenu plus généreux et a décidé d'ajouter 100€ au salaire brut de tous ses employés ! Combien de modifications devez-vous apporter à votre code, pour que ça fonctionne ?
6. Maintenant, votre client se rend compte qu'un `Commercial` ne peut pas être un simple commercial (donc ne peut pas être instancié en tant que tel), mais doit être distingué en tant que vendeur ou représentant. Ajoutez les deux classes correspondantes en faisant un héritage de `Commercial`. Vous ajouterez également à la classe `Commercial`, le code nécessaire afin que cette classe ne soit pas instanciable.
7. Pour terminer, faites en sorte que la méthode de calcul du salaire brut d'un vendeur soit la même que la méthode de calcul du salaire brut d'un commercial, alors que la formule de calcul du salaire brut des représentants soit la même que celle utilisée pour le salaire brut des techniciens. Ajoutez cette fonctionnalité dans votre application.

   Attention à ne pas dupliquer du code (principe DRY) et à ne pas modifier le code précédemment écrit ! Sinon la dette de votre logiciel va augmenter. ☺
8. Quels sont les avantages et inconvénients de votre approche ?
9. Question facultative : dessinez le diagramme de classes afin de mieux comprendre votre solution.