Développement Orienté Objets

IUT Montpellier-Sète – Département Informatique

- **Support de cours**
- **Enseignants:** Marin Bougeret, Gaëlle Hisler, Cyrille Nadal, Victor Poupet, Gilles Trombettoni, Petru Valicov
- Le forum Piazza de ce cours pour poser vos questions
- Email pour une question d'ordre privée concernant le cours.
- Le sujet du TP en format .pdf téléchargeable et imprimable.

Consignes (importantes pour la notation)

- Vous respecterez toutes les consignes indiquées dans le TP précédent
- Vous respecterez les conventions de nommage *Java* (vues en cours ou disponibles sur le site d'Oracle). Prêtez une attention particulière au respect des noms de classes, attributs et méthodes qui vous seront demandés.
- Dans ce TP, un principe important que vous devriez essayer de respecter dans votre code est le principe DRY.
- Pour chaque question nécessitant une vérification automatique, vous écrirez des tests unitaires, pour valider votre solution.
- **Vous nommerez les classes telles que demandé dans le sujet et respecterez les signatures des méthodes qui vous sont demandées. Sinon, votre code ne pourra pas être testé…**

TP 5 : Système de gestion des employés

Thème : Héritage - réutiliser, mais sans en abuser

Date limite de rendu de votre code sur le dépôt GitHub : **Dimanche 27 février 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…

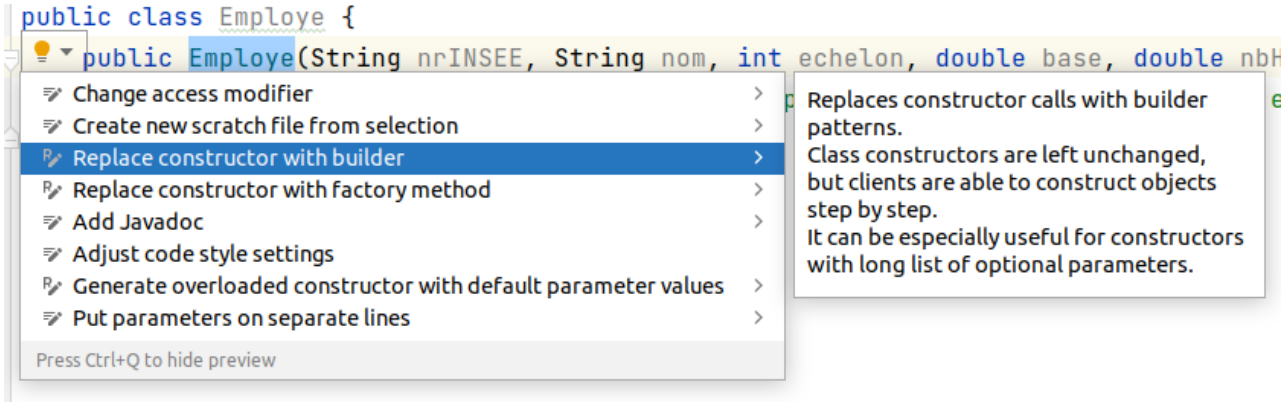
Dans tout ce TP, tous les attributs devraient être `private` (en particulier il est **très déconseillé** d'utiliser la visibilité `protected` pour les attributs).

Exercice 1

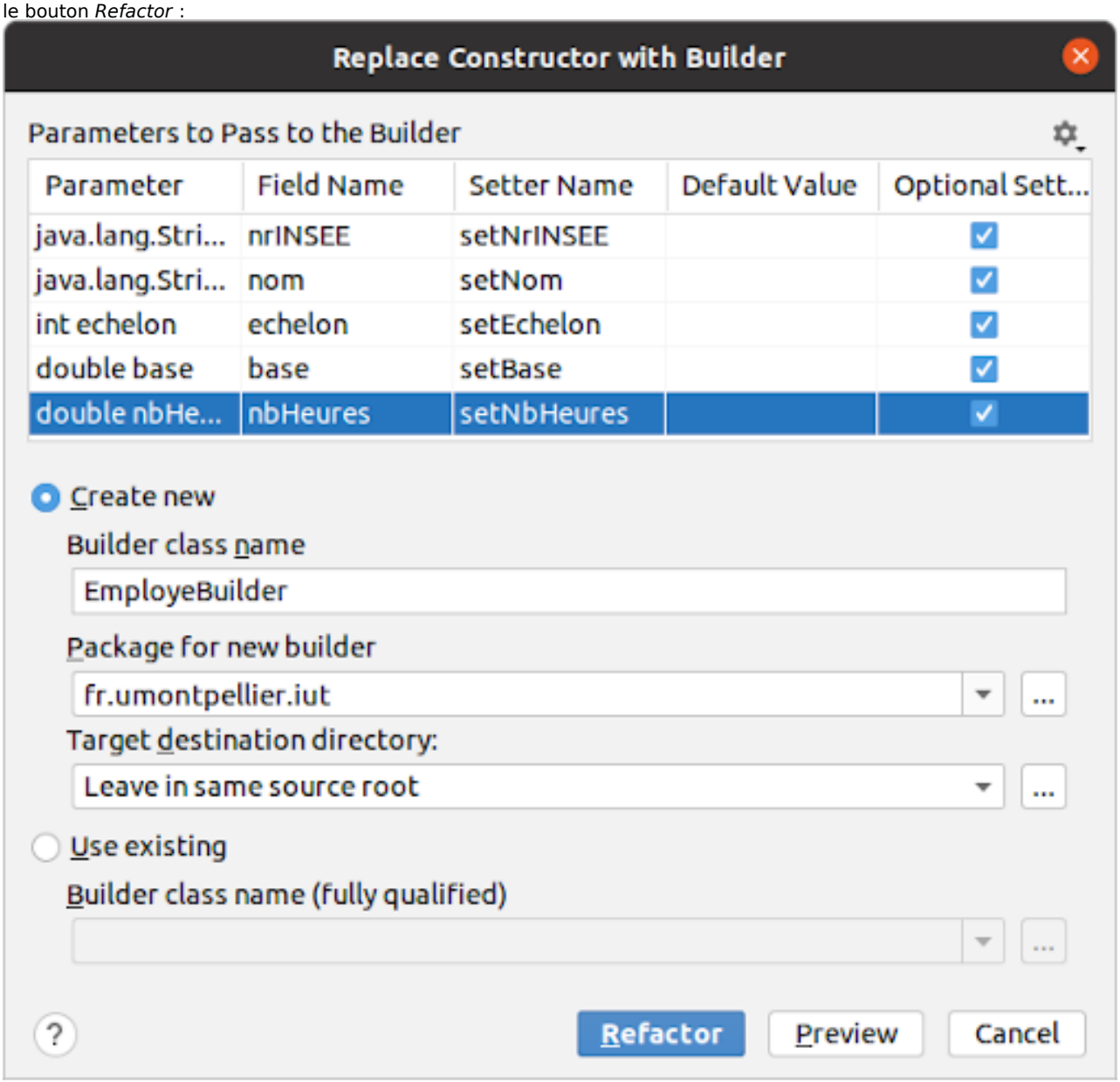
Vous êtes chargés de développer un système de gestion d'employés. Chaque employé est représenté par les données suivantes : numéro de sécurité sociale, nom, échelon (entier naturel). Pour pouvoir calculer le salaire brut et le salaire net d'un employé, deux attributs supplémentaires de type `double` sont nécessaires : `base` et `nbHeures`. Le constructeur de la classe `Employe` vous est donné, merci de garder **les paramètres dans cet ordre** :

```
public Employe(String nrINSEE, String nom, int echelon, double base, double nbHeures)
```

1. 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 → *Show Context Actions* → *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* :



- On peut continuer à refactoriser et rendre le Builder comme une classe interne statique de la classe `Employe` en faisant un glisser/déposer (drag & drop) de `EmployerBuilder` dans `Employe` dans la fenêtre de l'IDE.

  C'est sûrement possible de générer le builder dans d'autres IDE, on vous laisse vous documenter.

Important : générer le builder, ne veut pas dire changer le constructeur de la classe `Employe`... **il ne faut pas** changer la

signature du constructeur de `Employe` qui vous est donnée.

2. Le salaire brut de l'employé se calcule de la manière suivante : `base * nbHeures`. Le salaire net représentera toujours 80% du salaire brut. Ajoutez le code nécessaire (attributs et méthodes) pour intégrer ces fonctionnalités et proposez des méthodes respectives `getSalaireBrut()` et `getSalaireNet()`.

3. Redéfinissez la méthode `String toString()` dans la classe `Employe` pour afficher les informations concernant un employé (y compris ses salaires brut et net).

4. Si jamais votre client vous demande de changer (modifier donc) la formule de calcul du salaire brut et de la fixer à `base * nbHeures * 1.05`, combien de changements devriez-vous effectuer au minimum pour que votre programme continue de fonctionner correctement ?

   Remarque : comme convenu précédemment, dans ce qui suit, le salaire brut d'un employé restera toujours le même, à savoir `base * nbHeures`, donc aucun changement dans votre code pour la question 5.

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. N'oubliez pas les tests unitaires pour les exercices précédents !

Exercice 2

1. Avec le développement de l'entreprise, une séparation des traitements pour les différents types d'employés devient nécessaire. Il faut spécifier les cas des *Commerciaux*, *Fabricants*, et les autres employés qu'on appellera *Techniciens*.

   - La classe `Commercial` a comme attributs `chiffreAffaires` et `tauxCommission` (tous les deux de type `double`).
   - La classe `Fabricant` a comme attributs `nbUnitesProduites` et `tauxCommissionUnite` (type `int`et `double` respectivement).
   - La classe `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`.

   Voici la signature des constructeurs de ces classes :

```
public Commercial(String nrINSEE, String nom, int echelon, double base, double nbHeures, double chiffreA
public Fabricant(String nrINSEE, String nom, int echelon, double base, double nbHeures, int nbUnitesProd
public Technicien(String nrINSEE, String nom, int echelon, double base, double nbHeures)
```

   Ne générez pas des *builders* pour ces classes pour le moment.

2. Un commercial peut négocier des transactions avec la méthode `negocierTransaction(double sommeGagnée)`, elle incrémentera son chiffre d'affaires. Un fabricant fabrique des produits avec la méthode `void fabriquerProduits(int nbProduits)` ce qui incrémente son nombre d'unités produites. Un technicien effectue les autres tâches dans l'entreprise (méthode `void effectuerTacheTechnique()`) à travers un simple affichage d'un message approprié à la console. Toutes ces méthodes sont de type `void` et vous pouvez faire en sorte d'afficher un message approprié pour illustrer leur bon fonctionnement. Par exemple, la méthode `void negocierTransaction(double sommeGagnée)` pourra en plus 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 et du nombre de tâches effectuées. Plus concrètement, le résultat du retour de la fonction de calcul du salaire brut devra être égal 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 `getSalaireBrut()` 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. Déclarez un objet de type `Employe` et instanciez-le en tant que `Fabricant`. Observez le résultat de l'appel des méthodes `getSalaireBrut()` et `getSalaireNet()`. Est-ce que la méthode `fabriquerProduits(int nbProduits)` est accessible ? Expliquez en comparant avec le scénario où l'objet serait déclaré en tant que `Fabricant`.
6. Si jamais le patron devient plus généreux et décide d'ajouter une somme fixe de 100€ au salaire brut de tous ses employés, combien de modifications devez-vous apporter à votre code pour que cela fonctionne ? Attention, c'était une blague, le patron ne sera pas généreux, donc ne faites pas cette modification !

Exercice 3

1. 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 `Representant`. Un vendeur peut vendre des produits (méthode `void vendreProduit()`) et un représentant peut représenter l'entreprise auprès des différents clients (méthode `void representerEntreprise()`). Ajoutez les deux classes correspondantes en faisant un héritage de `Commercial`.

   Voici la signature des constructeurs de ces classes : `java public Vendeur(String nrINSEE, String nom, int echelon, double base, double nbHeures, double chiffreAffaires, double tauxCommission) public Representant(String nrINSEE, String nom, int echelon, double base, double nbHeures, double chiffreAffair double tauxCommission)`

   Pour rendre la classe `Commercial` non-instanciable il faut modifier sa déclaration en ajoutant le mot-clé `abstract` : `public abstract class`. Observez les changements à faire à l'utilisation des objets `Commercial`.
2. La méthode `void representerEntreprise()` de la classe `Representant` incrémente le nombre de représentations effectuées par ce salarié (un attribut de cette classe donc).
3. Implémentez les *builders* pour les classes `Technicien`, `Fabricant`, `Representant` et `Vendeur`.

   Remarque : observez la duplication de code entre les différentes classes *builders* (non-respect du principe DRY). Pour le moment, pour des raisons de facilité nous allons tolérer ce défaut et laisser les classes *builders* telles quelles. Dans quelques semaines, après avoir suffisamment avancé dans le cours, nous y reviendront pour améliorer. Pour les curieux : https://stackoverflow.com/questions/21086417/builder-pattern-and-inheritance Une explication approfondie et une solution sont également données dans *Effective Java* de J. Blosch, (3ème édition).
4. Pour terminer, faites en sorte que la méthode de calcul du salaire brut d'un vendeur soit *toujours* 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 *toujours* la même que celle utilisée pour le salaire brut des *techniciens* + le nombre de représentations * 123. 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. :smirk:
5. Quels sont les avantages et inconvénients de votre solution à la question 4 ?
6. Dessinez le diagramme de classes afin de mieux comprendre votre solution. Vous déposerez le diagramme sous forme d'image (.png ou .jpg) à la racine de votre dépôt Git.

   Note : les *builders* ne doivent pas faire partie de votre diagramme de classes.
7. **Bonus :** Si vous êtes vraiment en avance, essayez de proposer une solution qui évite la duplication de code entre les différentes classes de builder. Vous travaillerez dans un paquetage différent pour cela. Pour avoir une vraie solution flexible, il faudrait que :

   - Un `Employe` soit instantiable en tant que `Employe`, mais aussi en tant que `Fabricant`, `Representant`, etc.
   - Un `Commercial` soit instantiable en tant que `Representant` et `Vendeur`.
   - Un `Representant` puisse être instantialbe en tant que `Representant`, un `Technicien` en tant que `Technicien`, etc.
   - Les duplications de code entre les *setters* des différents builders soient supprimées.
   - Il n'y ait aucun *cast* dans votre code (conversion explicite d'un type à un autre, à la main).