COMPTE_RENDU SIVANANTHAM
TP1-C

EXERCICE 1:

2)

Bonus : Définit un certains bonus.
Employe: Définit un employé.
Payment: le coût des employé dans l'entreprise.

Le payement de l'entreprise est calculé en fonction de tout les employés(0 ou +) de celles ci. Le payement est calculé en fonction de son salaire et de son bonus (1 seul).

3)

Le champs amount dans Bonus est final car l'amount définit le bonus. Un nouvelle amount définit un nouveau Bonus. Le champs name dans Employee définit l'employé. Un nouveau name définit un nouvel Employee.

On vérifie si amount et salary sont positifs car on ne veut pas avoir des champs avec des valeur interdites par le problème.

Pour éviter cet petite duplication de code on peut créer une méthode privée en fesant du refactoring.

La méthode Objects.requireNonNull dans Employe et Payment permet de vérifier si les objets qu'on veut mettre dans l'instance de la classe ne sont pas null car les objets null sont interdit.

Si on renvoit la référence dans Payment.getAllEmployees, on a la possibilité de modifier l'objet de l'extérieur ce qui enfreint l'encapsulation.

Il n'y a pas de getters dans Employee et Bonus car les autres classes n'ont pas besoin de connaitre les champs de ces classes. C'est l'encapsulation.

5)

Le code pour calculer le payement il faudra : une méthode getCost() dans employé qui renverra le coût de l'employé en fonction de son bonus et de son salaire, une méthode getPayment() dans payement qui fera la somme des coûts des employés. Bien sur accédera au champs amount à partir de Bonus à partir d'Employee seulement si Bonus n'est pas null.

6)

Il faudrait créer une classe Student et une interface HasCost avec une méthode getCost(). Ensuite, on implémente HasCost dans Student et Employee et on modifier la classe payment pour qu'elle ne possède plus un ou plus Employe mais un ou plusieurs HasCost.

8)

On dit que l'héritage n'est pas un outil de design car il implique un fort couplage. Cela factorise bien le code, en revanche il complique la maintenance derrière car l'encapsulation est plus faible.

9)

Un petit changement comme celui ci peut entrainer beaucoup de modification dans notre architecture car les objets ont plusieurs fonction. Pour réduire ce genre de problème il faut réduire les fonctions (dans le sens fonctionnalités) des classes pour qu'elle soient le plus simple possible (délégation).