R5.A.08

TP 7

Exercice 1 : patron Builder

Commencer par relire le cours 6 sur le builder.
L'exemple proposé utilise le patron Builder pour construire des maisons en bois ou en brique.

L'exemple est presque complet pour les maisons en bois, donc il faut le terminer et surtout terminer le code pour les maisons en brique, notamment implémenter les classes suivantes :

```
// Similar structure for Brick family
public class BrickHouse extends House …
public class BrickFloor implements Floor …
public class BrickWalls implements Walls …
public class BrickRoof implements Roof …
```

La classe `HouseClient` est la classe lanceur de l'application.
Vous devrez obtenir la trace suivante à l'exécution :

```
>Building a wood house
>Finished building wood floor
>Finished building wood walls
>Finished building wooden roof
>
>Building a brick house
>Finished building brick floor
>Finished building brick walls
>Finished building brick roof
```

A faire

- Terminer le code et faire tourner l'application
- Faire la rétro-conception dans un diagramme de classes. Puis disposer au mieux les éléments et faire les modifications habituelles pour respecter la norme UML.
- Finalement indiquer le rôle des classes dans le pattern, soit sur le diagramme ou dans un texte à part.

Exercice 2 : patron Visitor (page 62)

L'objectif de cet exercice est d'utiliser le patron Visitor pour extraire des fonctionnalités d'une application qui en contient plusieurs.

Pour l'instant, notre application représente des groupes de sociétés qui gère des flottes de véhicules. Les sociétés sont avec ou sans filiales et leur fonctionnalités sont, d'une part, de calculer l'entretien pour les véhicules et, d'autre part, d'envoyer des propositions commerciales par email.

L'application est organisée sous la forme d'objets composés selon le pattern `Composite`.

Les deux sous-classes de la classe `Societe` possèdent deux méthodes de même nom : `calculeCoutEntretien` et `envoieEmailCommercial`.
Chacune de ces méthodes correspond à une même fonctionnalité mais dont l'implémentation est adaptée en fonction de la classe. De nombreuses autres fonctionnalités pourraient aussi être implémentées, par exemple le calcul du chiffre d'affaires d'un client etc..

Tant que le nombre de fonctionnalités reste faible l'approche de l'application actuelle est utilisable. Par contre, il faut trouver une autre approche si on est face à des classes contenant beaucoup de méthodes, difficiles à appréhender et à maintenir. On remarque aussi que ces méthodes n'ont pas de lien entre elles.

L'utilisateur du pattern **`Visitor`** s'impose. Il propose d'implémenter chaque fonctionnalité dans un visiteur séparé.

Voici quelques indications pour intégrer le pattern dans notre application /

- Chaque visiteur établit une fonctionnalité pour plusieurs classes en introduisant pour chaque classe une méthode dont on a décidé que le nom respecte une convention de nommage unique, à savoir **`visite`** suivi du nom de la classe : **`visiteSocieteMere`** `et` **`visiteSocieteSansFiliale`**
- Ensuite, le visiteur est transmis à la méthode **`accepteVisiteur`** des classes de société. Cette méthode appelle la méthode du visiteur correspondant à sa classe
- Si les objets sont composés alors leur méthode **`accepteVisiteur`** appelle la méthode `accepteVisiteur` de leurs composants (cas de la classe `SocieteMere`)

A faire

1. Réaliser le diagramme de classes de départ par rétro-conception du code.
2. Appliquer le pattern **`Visitor`** à l'application. Comme il y a deux fonctionnalités vous aurez deux classes de visiteurs : **`VisiteurCalculeEntretien`** et **`VisiteurMailingCommercial`** que vous relierez à une interface Visiteur.
3. Modifier le code de la classe **`Utilisateur`** pour qu'un visiteur soit créé (pour chaque fonctionnalité) et passé en paramètre d'une méthode `accepteVisiteur`.
4. Compiler et faire tourner la nouvelle application.
5. Faire la rétro-coonception pour obtenir le diagramme de classes final.