TD-TP de Compléments en Programmation Orientée Objet : Le patron "Monteur" (*builder pattern*)

Problème à résoudre : on veut construire des objets d'une certaine classe en se permettant de nombreuses options et paramètres, dont certains sont optionnels (ont une valeur par défaut). On veut, ce faisant, éviter de faire exploser le nombre de contsructeurs, et on veut qu'il soit impossible, à n'importe quel moment, qu'un objet de cette classe existe dans un état incohérent.

Exemple à modéliser : le curriculum vitae d'une personne. Pour simplifier, on s'occupe seulement de la partie "diplômes". Un CV contient ainsi les informations (attributs privés + getteurs) suivantes :

- `Bac bac` : l'information sur le bac obtenu, le cas échéant, `null` [1] si pas de bac.
- `DAEU daeu` : information sur le DAEU (Diplôme d'Accès aux Études Universitaires) obtenu, `null` si pas de DAEU
- `Licence licence` : information sur la licence obtenue, `null` si pas de licence
- `DiplomeInge dInge` : information sur le diplôme d'ingénieur obtenu, `null` si pas de diplôme d'ingénieur
- `Master master` : information sur le master obtenu, `null` si pas de master
- `Doctorat doctorat` : information sur le doctorat obtenu, `null` si pas de doctorat.

On suppose que les classes `Bac`, `DAEU`,`Licence`, `DiplomeInge` et `Doctorat` étendent toutes la classe `Diplome` [2] contenant les informations sur l'intitulé, l'annéée d'obtention et la mention obtenue au diplôme :

```java
public abstract class Diplome {
    public final String intitule;
    public final Mention mention;
    public final int annee;

    public Diplome(String intitule, Mention mention, int annee) {
        this.intitule = intitule;
        this.mention = mention;
        this.annee = annee;
    }
}

public final class Bac extends Diplome {
    public Bac(String intitule, Mention mention, int annee) {
        super(intitule, mention, annee);
    }
}

// et pareil pour DAEU, Licence, DiplomeInge, Master, Doctorat...

// le type Mention est défini comme suit :
public enum Mention { PASSABLE, ASSEZ_BIEN, BIEN, TRES_BIEN, FELICITATIONS; }
```

Les contraintes à respecter pour qu'un CV soit valide sont les suivantes :

- Pour avoir une licence ou un diplôme d'ingénieur, il faut avoir eu le bac ou un DAEU.
- Pour avoir un master, il faut avoir eu une licence ou un diplôme d'ingénieur.

1. Pour des raisons de concision du sujet, nous y utilisons `null`, pour signifier une absence de valeur.
En général, utiliser `null` pour ce cas n'est en réalité pas une bonne pratique, car `null` peut vouloir dire plein de choses différentes, d'une part, et d'autre part parce qu'on risque de propager cette valeur bien plus loin dans l'exécution, là où on l'utilisera en ayant oublié qu'elle est susceptible d'être `null` (et là, probablement, on tentera d'appeler dessus une méthode, ce qui provoquera un `NullPointerException`).
Pour mieux faire, renseignez-vous sur la classe `Optional<T>`, introduite dans Java 8.

2. On peut aussi faire sans héritage en mettant le contenu de `Diplome` directement dans les différentes classes de diplômes.

— Pour avoir un doctorat, il faut avoir eu un master.

Exercice 1 : À l'aide de constructeurs

Écrire la classe `CurriculumVitae` munie de constructeurs prenant en paramètre les diplômes obtenus. Permettre de multiples versions (surcharges) du constructeur, de telle sorte qu'il soit possible de ne passer que les diplômes effectivement obtenus en paramètre. L'idée est de ne jamais avoir à passer la valeur `null` en paramètre au constructeur [3].

Combien de constructeurs avez-vous écrits ? Supposez qu'on ajoute aussi un champ pour le DAEU (Diplôme d'Accès aux Études Universitaires, équivalent au bac), combien en faudrait-il ?

Exercice 2 : À l'aide de setteurs "optimistes"

Les constructeurs ne nous ayant pas pleinement satisfaits, optons pour l'approche suivante, qui résoud le problème soulevé à l'exercice précédent :

— Ne garder qu'un seul constructeur, sans paramètre, laissant les attributs initialisés à leur valeur par défaut (pas de diplôme).

— Écrire un "setteur" pour chaque attribut, avec la signature suivante : `public void setTruc(Truc truc)`, ayant pour effet d'affecter la valeur `truc` à l'attribut correspondant, sans vérifier la cohérence du CV.

Quel problème peut se poser avec cette approche ? Par exemple, si on exécute :

```
1 CurriculumVitae cv = new CurriculumVitae();
2 cv.setDoctorat(new Doctorat("Xénobiologie", Mention.TRES_BIEN, 2022);
```

alors, est-ce que le CV obtenu est cohérent ?

Exercice 3 : À l'aide de setteurs "pessimistes"

Pour faire en sorte que le scénario ci-dessus ne puisse pas se produire, nous décidons de programmer les setteurs avec une signature modifiée : `public boolean setTruc(Truc truc)`, qui ne font la modification que si le CV modifié est cohérent. Dans ce cas, ils retournent `true`. Dans le cas contraire, si la modification n'est pas autorisée, elle ne sera pas faite, et le setteur retourne `false`.

1. Est-ce que cette façon de faire résoud le problème posé à l'exercice précédent ?
2. Quel nouveau problème cette approche pose-t-elle ?
   (Imaginez des cas aux contraintes un peu plus extrêmes :
   — Au lieu de la classe `CurriculumVitae` on programme la classe `CarreMagique` contenant un tableau carré d'entiers naturels, dont la contrainte de cohérence est que, à tout instant, toutes les lignes et toutes les colonnes ont la même somme.
   — Ou bien, même chose pour la classe `Sudoku` qui, par spécification, ne pourrait représenter qu'une grille de Sudoku résolue.
   — Ou bien, toute classe dont les instances contiendraient des données telles que, pour chaque élément de donnée, sa cohérence dépendrait de tous les autres éléments.
   À supposer que l'appel à un setteur n'ait aucun effet si la modification proposée mène l'objet vers un état incohérent, arriverait-on, à l'aide de leurs setteurs, à modifier une instance d'une des classes ci-dessus pour passer d'un état cohérent à un autre ?)

3. L'utilisation de la valeur `null` comme entrée ou sortie normale dans l'interface publique d'une classe est considérée comme une mauvaise pratique.

3. Par ailleurs, que les setteurs soient "optimistes" ou "pessimistes", si on décide que la classe `CurriculumVitae` est immuable [4], est-ce que cette approche à des chances de fonctionner ?

Exercice 4 : Le patron "monteur"

Introduisons une nouvelle technique, qui contourne tous les écueils repérés dans les exercices précédents.

L'idée est la suivante : on s'aide d'une classe auxiliaire (le monteur ou *builder*), dont les instances peuvent être initialisées de façon souple, en plusieurs étapes, à l'aide de setteurs. La classe principale (dont on veut construire des instances) sera dépourvue de setteurs, et ses objets seront initialisés en un seul appel à son constructeur, prenant en paramètre une instance de *builder*.

Exemple :

```
1
2  public class Point {
3      public final int x, y;
4      public Point(PointBuilder builder) {
5          x = builder.x; y = builder.y;
6      }
7  }
8
9  public class PointBuilder {
10     public int x, y;
11 }
```

À faire : Transformez l'exemple pour lui faire adopter le patron "monteur" (pour cela, créez une classe auxiliaire `CVBuilder`). Attention, contrairement à l'exemple ci-dessus il faudra, dans le constructeur de `CurriculumVitae`, vérifier la cohérence des données fournies par le monteur. En cas d'incohérence, lever une exception (insérer l'instruction `throw new IllegalArgumentException();` dans la branche de code concernée).

Exercice 5 : Un peu de toilettage

L'exercice précédent montre le principe du patron "monteur", mais cette implémentation a quelques défauts (réparables) :

— La classe `CVBuilder` ne devrait pas être manipulée de façon si "ostensible". Ce qui intéresse l'utilisateur, c'est la classe `CurriculumVitae`.

— L'encapsulation est mauvaise, on accède aux détails internes de `CVBuilder` (accès direct aux attributs).

— On a besoin de plusieurs instructions pour créer une instance de `CurriculumVitae`. Exemple :

```
1  CVBuilder builder = new CVBuilder();
2  builder.bac = new Bac("S", Mention.BIEN, 2015);
3  builder.licence = new Licence("SVT", Mention.ASSEZ_BIEN, 2018);
4  CurriculumVitae cvJeanJacques = new CurriculumVitae(builder);
```

En pratique, on aimerait une implémentation un peu plus propre, donnant, d'une part, une visibilité minimale à la classe du monteur et ses attributs, et fournissant d'autre part des méthodes pratiques permettant une construction à la syntaxe "abrégée". Exemple de construction de CV :

4. C'est-à-dire, dont les instances ne sont pas modifiables. Notamment, les attributs d'instance sont tous `final`. Programmer en utilisant de tels objets facilite le déboguage et donne souvent des garanties de robustesse.

```
1  CurriculumVitae cvJeanJacques = CurriculumVitae.builder()
2        .bac(new Bac("S", Mention.BIEN, 2015))
3        .licence(new Licence("SVT", Mention.ASSEZ_BIEN, 2018))
4        .build();
```

Notez l'absence de référence explicite à la classe `CVBuilder`. La méthode `builder()` est une méthode statique de `CurriculumVitae` appelant le constructeur de `CVBuilder` ; les appels intermédiaires sont juste des setteurs de la classe `CVBuilder`, qui retournent `this` (au lieu de rien) et la méthode `build()` appelle le constructeur de `CurriculumVitae` et retourne l'instance construite.

À faire : Transformez votre implémentation de `CurriculumVitae` et `CVBuilder` afin de la toiletter comme indiqué et de rendre possible l'invocation ci-dessus. Ajoutez des `toString()` dans toutes vos classes et testez sur quelques exemples.

On pourra ajouter une méthode booléenne à `CVBuilder` pour vérifier la cohérence de ses données avant-même d'appeler `build()`.

Remarque : les constructeurs de `CVBuilder` et `CurriculumVitae` n'ont plus aucun intérêt à rester publics[5] : il est, en effet, plus pratique, et équivalent, d'appeler, respectivement, les méthodes `builder()` et `build()`.

5. On pourra leur donner une visibilité *package-private* (en mettant les 2 classes dans le même *package*). Variante avancée : visibilité `private`, en écrivant `CVBuilder` en tant que classe membre statique de `CurriculumVitae`.