TD n° 2 : Objets, classes, encapsulation

I) Retour sur l'exercice 5 du TP1

Ce qu'on attendait pour la classe immuable (fin de la question 3) :

- Des attributs parties réelles et imaginaires **`private final`**, non statiques.
- Des attributs `I`, `ZERO` et `UN` **`public final static`** (il s'agit de constantes).
- La méthode `somme` peut s'écrire

```java
// appel du type Complexe c = a.somme(b);
public Complexe somme(Complexe autre) {
    return new Complexe(this.re + autre.re, this.im + autre.im);
}
```

ou (mais c'est moins bien)

```java
// appel du type Complexe c = Complexe.somme(a,b);
public static Complexe somme(Complexe arg1, Complexe arg2) {
    return new Complexe(arg1.re + arg2.re, arg1.im + arg2.im;
}
```

Le changement de nom s'explique par le changement de sémantique. Il faut toujours privilégier les méthodes non statiques si on n'a pas de bonne raisons de faire autrement.

Ce qu'on attendait pour la classe mutable (question 5) :

- Les attributs parties réelles et imaginaires ne sont plus **`final`**, ils sont toujours non statiques, et ils doivent être **`private`**.
- Les attributs `I`, `ZERO` et `UN` doivent être supprimés car sinon, il serait possible d'écrire ZERO.setIm(3.57),!! On peut les remplacer par des fabriques statiques (le résultat sera modifiable, mais l'appel à la fabrique statique donnera toujours un nouveau complexe qui aura la bonne valeur).
- La méthode `plus` peut s'écrire

```java
// appel du type a.plus(b);
public void plus(Complexe autre) {
    return this.re = this.re + autre.re;
    this.im = this.im + autre.im);
}
```

Le changement de nom s'explique par le changement de sémantique.

Si vous n'avez pas fait ça, reprenez votre code. Pour gagner du temps, vous pouvez choisir de ne traiter que l'addition et la soustraction, que ZERO (ou la fabrique statique correspondante).

II) Programmer

Les "patterns" sont des réponses stylistiques à certains problèmes de programmation qui sont répertoriés dans la littérature car ils sont rencontrés assez fréquemment. Ces réponses doivent faire partie de votre culture générale, vous pourrez ainsi vous y référer pour définir votre propre style.

Le pattern présenté dans cette section vient, avec une certaine élégance, résoudre un problème spécifique : nous abordons le builder-pattern (ou patron monteur en français).

Problème général à résoudre : on veut construire des objets d'une certaine classe en se permettant de nombreuses options et paramètres, dont certains sont optionnels ou ont une valeur

par défaut. On veut, ce faisant, éviter de faire exploser le nombre de constructeurs, et on veut qu'il soit impossible, à n'importe quel moment, qu'un objet de cette classe existe dans un état incohérent.
Le problème sera illustré sur un exemple concret. Tout d'abord en se lançant dans des résolutions triviales pas tout à fait satisfaisantes, puis on présentera la solution "pattern-builder".
Problème concret à modéliser : on veut définir un objet pour le curriculum vitae d'une personne. Pour simplifier, on s'occupe seulement de la partie qui énumère ses "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'année 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 auparavant.
- Pour avoir un master, il faut avoir eu une licence ou un diplôme d'ingénieur auparavant.
- Pour avoir un doctorat, il faut avoir eu un master auparavant.

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 faire les exercices suivants, ne vous occupez pas du DAEU, mais essayez de voir en quoi son ajout complexifierait les choses suivant les approches proposées.

Exercice 1 : Première approche - pattern Telescoping Constructor

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].
On utilisera le pattern Telescoping Constructor, qui dans le cours est utilisé pour la classe `NutritionFactsT`.
Levez une exception en cas d'incohérence. Il est possible de gérer les exceptions uniquement dans le constructeurs avec tous les arguments.
Combien de constructeurs avez-vous pu écrire avant de vous fatiguer ? Combien au faudrait-il si le DEUG était à nouveau un diplôme délivré après 2 années d'université ?

Exercice 2 : Seconde approche - Java Beans ou 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 : Troisième approche - 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.

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.

  - 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. 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*.
Illustration triviale d'un builder :

```
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);
```

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.

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 :

```
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. `CVBuilder` deviendra une classe interne statique. 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()`.

III) Pourquoi faire simple quand on peut faire compliqué[6] : la réflexion

Cet exercice est **complètement facultatif**, il s'adresse aux étudiants rapides qui ont peur de s'ennuyer.

La façon de programmer que nous vous montrons n'est pas du tout recommandée, sauf cas très particuliers. Inutile de l'utiliser dans le projet, cela ne vous rapportera ausun point, sauf bien sûr si vous arrivez à justifier cette utilisation.

Le but de cet exercice est de vous montrer un peu comment marche la machine java.

Si vous voulez en savoir plus : `https://docs.oracle.com/javase/tutorial/reflect/` et `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/package-summary.html`.

Exercice 6 : Version de base

On désire faire une classe Calcul qui lorsqu'elle est lancée en ligne de commande avec comme argument une fonction mathématique et une valeur double, calcule le résultat de cette fonction avec comme paramètre la valeur puis l'affiche. Par exemple :

```
> java Calcul cos 1.1
0.4535961214255773
```

L'idée est de faire ça sans avoir besoin de faire une liste des fonctions unaires de la classe `Math` qui prennent un double en argument et retournent un double. Nous allons vous guider étape par étape.

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`.

6. proverbe shadok

1. Tout d'abord, nous vous rappelons que les arguments de la ligne de commande `java Calcul cos 1.1`, "cos" et "1.1" sont respectivement dans `args[0]` et `args[1]` où `args` est l'argument du main (`.. main(String[] args)`).
2. Pour simplifier, dans un premier temps, indiquez `throws Exception` pour le `main`, vous traiterez les exceptions plus tard, s'il vous reste du temps.
3. Il vous faudra un `import java.lang.reflect.*;`
4. La première chose à faire est de récupérer l'objet de type `Class` qui correspond à la classe `Math`. Pour ce faire, on utilise la méthode `Class.forName(..)` : en argument il faut lui donner le nom complet de la classe : "java.lang.Math" et pour typer le résultat vous pouvez utiliser `Class<?>`.
5. pour récupérer, l'objet de type `Class` qui correspond au type primitif `double`, la syntaxe est différente : c'est `double.class`.
6. Maintenant, il faut récupérer la méthode qui correspond à la fonction voulue (dans notre exemple, `cos`). C'est la méthode `getMethod` de `Class` qu'il faut utiliser. Regardez la doc de l'API java, vous devriez y arriver.
7. Il faut maintenant appliquer la fonction à la valeur donné en argument. Bien sûr, il faudra faire la conversion de la chaîne de caractères vers un `double`, mais surtout, il faut utiliser la méthode `invoke` sur la méthode récupérée. Faites attention au fait que les méthodes de `Math` sont statiques, donc le premier argument de `invoke` est `null`.
8. Ensuite, il faudra convertir l'objet récupéré en double et l'afficher, et c'est fini.

Exercice 7 : Pour aller plus loin

Vous pouvez au choix :

- Traitez correctement les options.
- Expérimenter d'autres possibilités de la réflexion : par exemple traiter le cas des constantes `E` et `PI` de la classe `Math`.