Université Paris Cité

Compléments en Programmation Orientée Objet
TD n° 2 : Objets, classes, encapsulation
(Correction)

1 Questions de cours

Exercice 1 : private mais pour qui ?

Est-ce que le programme suivant compile et si oui qu'est-cee que affichent les `printf` sur les lignes : 43, 46, 50, 52 ?

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public static void modifPoint(Point target, Point source) {
        target.x = source.y;
        target.y = source.x;
    }

    //peut-on accéder aux attributs privés d'un autre point ?
    public void addThis( Point other ){
        this.x += other.x ;
        this.y += other.y ;
    }
   //peut-on modifier les attriburs privés d'un autre point ?
    public void addOther(Point other) {
        other.x += this.x;
        other.y += this.y;
    }

    @Override
    public String toString() {
        return String.format("(%d,%d)", x, y);
    }
}

public class Main {
    public static void main(String[] args) {
        Point a = new Point(3, 5);
        Point b = new Point(11, 22);
        System.out.printf( "a=%s\n",a );
        //modifier a par une méthode static ?
        Point.modifPoint(a, b);
        System.out.printf("a=%s\n", a );
        Point o = new Point( -5, -5);
        //modifier o par b ?
        b.addOther(o);
        System.out.printf("o=%s\n",o);
        b.addThis(a);
        System.out.printf("b=%s\n", b);
    }
}
```

Correction : L'affichage :

```
a=(3,5)
a=(22,11)
o=(6,17)
b=(33,33)

```

Conclusion : encapsulation par **`private`**) est vis-àvis d'autres classes et non pas vis-à-vis d'autres objets de la même classe ni même pour les méthodes **`static`** de la classe.

Exercice 2 : Encapsulation

Examinez le programme suivant et dites ce qui se passe ? Est-ce que le comportement est conforme aux commentaires du `main` ? Que faut-il corriger et comment ?

```java
class Personne {
    private String nom;
    private final int numSecu;

    public Personne(String nom, int numSecu) {
        this.nom = nom;
        this.numSecu = numSecu;
    }

    public void changeNom(String nom) {
        this.nom = nom;
    }

    @Override
    public String toString() {
        return String.format("%s %d", nom, numSecu);
    }
}

class Cours {
    private String nom;
    private HashSet<Personne> inscrits;

    public Cours(String nom, HashSet<Personne>
     inscrits) {
        this.nom = nom;
        this.inscrits = inscrits;
    }

    //return false si pas dans la liste
    //   des inscrits au départ
    public boolean exclut(Personne p) {
        return inscrits.remove(p);
    }

    //return false si déjà  dans liste
    //   des inscrits au départ
    public boolean inscrit(Personne p) {
        return inscrits.add(p);
    }

    @Override
    public String toString() {
        StringBuilder builder = new
         StringBuilder().append(nom);
        for (Personne p : inscrits) {
            builder.append("\n").append(p);
        }
        return builder.toString();
    }
}

public class Main {
    public static void main(String[] args) {
        //on crée un certain nombre de personnes:
        Personne p1 = new Personne("Adèle", 1254);
        Personne p2 = new Personne("Brian", 1287);
        Personne p3 = new Personne("Coralie", 2546);
        Personne p4 = new Personne("Désiré", 2546);
        //on crée un groupe de copains:
        HashSet copains = new HashSet<Personne>();
        copains.add(p1);
        copains.add(p2);
        copains.add(p3);
        //pour ce groupe, on crée 2 cours:
        Cours couture = new Cours("Couture", copains);
        Cours karate = new Cours("Karate", copains);
        //Coralie est exclue du cours de Couture
        couture.exclut(p3);
        //Désiré s'inscrit à  celui de Karaté
        karate.inscrit(p4);
        //Adèle change de prénom, parce que
        //  l'ancien ne lui plaisait pas
        p1.changeNom("Adeline");
        System.out.printf("%s\n\n%s", couture,
     karate);
    }
}
```

Remarque : En général, pour des raisons de persistence des données, ce genre de données a vocation à être géré par une base de données en interfaçage éventuel avec un langage de programmation.

Correction : Comportement observé :

```
Couture
Adeline 1254
Brian 1287
Désiré 2546

Karate
Adeline 1254
Brian 1287
Désiré 2546
```

Il faut faire une copie du `HashSet` à la création de cours :

```java
public Cours(String nom, HashSet<Personne> inscrits) {
     this.nom = nom;
     (this.inscrits = new HashSet<>( )).addAll(inscrits) ;
}
```

Par contre le changement de nom ne pose pas de problème.

2 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 :

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 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.

Exercice 3 : Première approche - 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]. Levez une exception en cas d'incohérence.
Combien de constructeurs avez-vous pu écrire avant de vous fatiguer ? Combien en faudrait-il si le DEUG était à nouveau un diplôme délivré après 2 années d'université ?

Correction :

```
1           public final class CurriculumVitae {
2               private final Bac bac;
3               private final Licence licence;
4               private final DiplomeInge dInge;
5               private final Master master;
6               private final Doctorat doctorat;
7
8               public CurriculumVitae() {
9                   this(null, null, null, null, null);
10              }
11
12              public CurriculumVitae(Bac bac) {
13                  this(bac, null, null, null, null);
14              }
15
16              public CurriculumVitae(Bac bac, Licence licence) {
17                  this(bac, licence, null, null, null);
18              }
19
20              public CurriculumVitae(Bac bac, DiplomeInge dInge) {
21                  this(bac, null, dInge, null, null);
22              }
23
24              public CurriculumVitae(Bac bac, Licence licence, DiplomeInge dInge) {
25                  this(bac, licence, dInge, null, null);
26              }
27
28              public CurriculumVitae(Bac bac, Licence licence, Master master) {
29                  this(bac, licence, null, master, null);
30              }
31
32              public CurriculumVitae(Bac bac, DiplomeInge dInge, Master master) {
33                  this(bac, null, dInge, master, null);
34              }
35
36              public CurriculumVitae(Bac bac, Licence licence, DiplomeInge dInge, Master master) {
37                  this(bac, licence, dInge, master, null);
38              }
39
40              public CurriculumVitae(Bac bac, Licence licence, Master master, Doctorat doctorat) {
41                  this(bac, licence, null, master, doctorat);
42              }
43
44              public CurriculumVitae(Bac bac, DiplomeInge dInge, Master master, Doctorat doctorat)
    {
45                  this(bac, null, dInge, master, doctorat);
46              }
47
48              public CurriculumVitae(Bac bac, Licence licence, DiplomeInge dInge, Master master,
    Doctorat doctorat) {
```

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.

```
49                  if ((bac == null && licence != null)
50                  || (bac == null && dInge != null)
51                  || (licence == null && dInge == null && master != null)
52                  || (master == null && doctorat != null))
53                  throw new IllegalArgumentException();
54                  this.bac = bac;
55                  this.licence = licence;
56                  this.dInge = dInge;
57                  this.master = master;
58                  this.doctorat = doctorat;
59              }
60
61              public Bac getBac() {
62                  return bac;
63              }
64
65              public Licence getLicence() {
66                  return licence;
67              }
68
69              public DiplomeInge getdInge() {
70                  return dInge;
71              }
72
73              public Master getMaster() {
74                  return master;
75              }
76
77              public Doctorat getDoctorat() {
78                  return doctorat;
79              }
80          }
81
```

Rien que pour cet exemple-là, il faut 11 constructeurs, 31 en ajoutant le DAEU (en supposant qu'on puisse avoir à la fois un bac et un DAEU)... et ça pourrait être encore pire, si on enlevait les dépendances entre les diplômes : il en faudrait 64 !

Exercice 4 : Seconde approche - 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 ?

Correction :

```
1           public final class CurriculumVitae {
2               private Bac bac;
3               private Licence licence;
4               private DiplomeInge dInge;
5               private Master master;
6               private Doctorat doctorat;
7
8               public Bac getBac() {
9                   return bac;
```

```
10              }
11
12              public void setBac(Bac bac) {
13                  this.bac = bac;
14              }
15
16              public Licence getLicence() {
17                  return licence;
18              }
19
20              public void setLicence(Licence licence) {
21                  this.licence = licence;
22              }
23
24              public DiplomeInge getdInge() {
25                  return dInge;
26              }
27
28              public void setdInge(DiplomeInge dInge) {
29                  this.dInge = dInge;
30              }
31
32              public Master getMaster() {
33                  return master;
34              }
35
36              public void setMaster(Master master) {
37                  this.master = master;
38              }
39
40              public Doctorat getDoctorat() {
41                  return doctorat;
42              }
43
44              public void setDoctorat(Doctorat doctorat) {
45                  this.doctorat = doctorat;
46              }
47          }
48
```

Clairement, avec cette implémentation, tout est permis dans le CV, il n'y a plus de règles. Ainsi, le CV obtenu à la fin de l'exemple donné n'est pas cohérent : cette personne a eu un doctorat sans avoir d'autres diplômes avant.

Exercice 5 : 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 ?

   Correction : Oui, tout à fait on peut faire tous les tests nécessaires dans les setteurs.

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 ?)

Correction : Le nouveau problème qui se pose, c'est que pour modifier un CV cohérent vers un autre CV cohérent, il se peut que plusieurs modifications soient nécessaires, et que les CVs des étapes intermédiaires ne soient pas cohérents.
Dans l'exemple du CV, il se trouve qu'on peut toujours trouver une séquence de transformations dont toutes les étapes sont cohérentes (par exemple : supprimer tous les diplômes un à un en commençant par celui de niveau le plus haut, puis les remettre un à un, en commençant par le diplôme de niveau le plus bas.), mais c'est très fastidieux. Dans d'autres exemples (comme `CarreMagique`), il se peut qu'un tel chemin n'existe pas du tout.

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 ?

Correction : Aucune chance : les setteurs servent à modifier l'objet. Il y aurait contradiction.

Correction : Pour référence, voici la nouvelle version de la classe `CurriculumVitae` (ce n'était pas demandé).

```
public class CurriculumVitae {
    private Bac bac;
    private Licence licence;
    private DiplomeInge dInge;
    private Master master;
    private Doctorat doctorat;

    public Bac getBac() {
        return bac;
    }

    public boolean setBac(Bac bac) {
        if (licence == null && dInge == null || bac != null) {
            this.bac = bac;
            return true;
        } else return false;
    }

    public Licence getLicence() {
        return licence;
    }

    public boolean setLicence(Licence licence) {
        if ((master == null || dInge != null || licence != null) // dépendances avant
        && (bac != null || licence == null)) { // dépendance arrière
            this.licence = licence;
            return true;
        } else return false;
    }

    public DiplomeInge getdInge() {
        return dInge;
    }

    public boolean setdInge(DiplomeInge dInge) {
        if ((master == null || dInge != null || licence != null) // dépendances avant
        && (bac != null || dInge == null)) { // dépendance arrière
            this.dInge = dInge;
            return true;
```

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.

```
                } else return false;
            }

            public Master getMaster() {
                return master;
            }

            public boolean setMaster(Master master) {
                if ((master == null || licence != null || dInge != null) && (master != null ||
doctorat == null)) {
                    this.master = master;
                    return true;
                } else return false;
            }

            public Doctorat getDoctorat() {
                return doctorat;
            }

            public boolean setDoctorat(Doctorat doctorat) {
                if (doctorat == null || master != null) {
                    this.doctorat = doctorat;
                    return true;
                } else return false;
            }
        }

```

Exercice 6 : 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 :

```

        public class Point {
            public final int x, y;
            public Point(PointBuilder builder) {
                x = builder.x; y = builder.y;
            }
        }

        public class PointBuilder {
            public int x, y;
        }

```

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).

Correction :

```
            public class CVBuilder {
                Bac bac;
                Licence licence;
                DiplomeInge dInge;
                Master master;
                Doctorat doctorat;
```

```
7               }
8
9               public class CurriculumVitae {
10                  private final Bac bac;
11                  private final Licence licence;
12                  private final DiplomeInge dInge;
13                  private final Master master;
14                  private final Doctorat doctorat;
15
16                  public CurriculumVitae(CVBuilder builder) {
17                      if ((builder.bac == null && builder.licence != null)
18                      || (builder.bac == null && builder.dInge != null)
19                      || (builder.licence == null && builder.dInge == null && builder.master !=
    null)
20                      || (builder.master == null && builder.doctorat != null))
21                      throw new IllegalArgumentException();
22                      this.bac = builder.bac;
23                      this.licence = builder.licence;
24                      this.dInge = builder.dInge;
25                      this.master = builder.master;
26                      this.doctorat = builder.doctorat;
27                  }
28
29                  public Bac getBac() {
30                      return bac;
31                  }
32
33                  public Licence getLicence() {
34                      return licence;
35                  }
36
37
38                  public DiplomeInge getdInge() {
39                      return dInge;
40                  }
41
42                  public Master getMaster() {
43                      return master;
44                  }
45
46                  public Doctorat getDoctorat() {
47                      return doctorat;
48                  }
49              }
50
51
```

Exercice 7 : 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);
5
```

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 :

```
CurriculumVitae cvJeanJacques = CurriculumVitae.builder()
.bac(new Bac("S", Mention.BIEN, 2015))
.licence(new Licence("SVT", Mention.ASSEZ_BIEN, 2018))
.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()`.

Correction :

```
public class CurriculumVitae {
    private final Bac bac;
    private final Licence licence;
    private final DiplomeInge dInge;
    private final Master master;
    private final Doctorat doctorat;

    private CurriculumVitae(Builder builder) {
        this.bac = builder.bac;
        this.licence = builder.licence;
        this.dInge = builder.dInge;
        this.master = builder.master;
        this.doctorat = builder.doctorat;
    }

    public static Builder builder() {
        return new Builder();
    }

    public Bac getBac() {
        return bac;
    }

    public Licence getLicence() {
        return licence;
    }


    public DiplomeInge getdInge() {
        return dInge;
    }

    public Master getMaster() {
        return master;
    }

    public Doctorat getDoctorat() {
        return doctorat;
    }

    public static class Builder {
        private Bac bac;
        private Licence licence;
```

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`.

```
44                      private DiplomeInge dInge;
45                      private Master master;
46                      private Doctorat doctorat;
47
48                      public CurriculumVitae build() {
49                          if (!estCoherent()) throw new IllegalStateException();
50                          return new CurriculumVitae(this);
51                      }
52
53                      public Builder bac(Bac bac) {
54                          this.bac = bac;
55                          return this;
56                      }
57
58                      public Builder licence(Licence licence) {
59                          this.licence = licence;
60                          return this;
61                      }
62
63                      public Builder dInge(DiplomeInge dInge) {
64                          this.dInge = dInge;
65                          return this;
66                      }
67
68                      public Builder master(Master master) {
69                          this.master = master;
70                          return this;
71                      }
72
73                      public Builder doctorat(Doctorat doctorat) {
74                          this.doctorat = doctorat;
75                          return this;
76                      }
77
78                      public boolean estCoherent() {
79                          return !((bac == null && licence != null)
80                          || (bac == null && dInge != null)
81                          || (licence == null && dInge == null && master != null)
82                          || (master == null && doctorat != null));
83                      }
84                  }
85              }
86
```