

JAVA SE - Encapsulation et Bean

Du bon usage de l'encapsulation et des conventions de Bean

Le principe d'encapsulation veut que l'on distingue, d'une part, ce qui est sous la responsabilité du programmeur en charge de la classe (ce qui est `private`) et, d'autre part, les services que ce programmeur offre à d'autres codes "client" (ce qui est `public`).

Il y a plusieurs raisons d'opérer ainsi:

- Un programmeur est en charge du code d'une classe : il offre aux autres programmeurs une garantie de service. Le code de la classe obéit à des règles métier, il doit être cohérent et les programmeurs "clients" doivent pouvoir lui faire confiance. Il est donc exclu que d'autres programmeurs puissent intervenir de manière inappropriée et puisse induire des modifications de l'objet qui ne respectent pas ces règles métier.
- Le code de la classe peut évoluer dans le temps (par exemple en modifiant les types des variables membres). Il n'est pas acceptable que ces modifications se répercutent sur des codes clients (ceci est surtout vrai quand on publie des composants qui sont utilisés par des programmeurs que nous ne connaissons pas).

Une convention classique en Java est de mettre en place des accesseurs/mutateurs.

Pour une variable membre :

```
private Type variable ;
```

On pourra avoir :

```
public Type getVariable() { // code
```

```
public void setVariable(Type valeur) { // code
```

Même si on aboutit à des méthodes qui ont un curieux nom mélangeant l'anglais et le français on a ici une convention de nommage qui a son utilité.

Il faut toutefois appliquer ces conventions avec discernement.

Supposons que nous ayons une classe comme celle-ci :

```
public class Client {
    private String référenceUnique ;
    private String nom ;
    private String prénom ;
    private Adresse adresse ;

    private int pointsFidélité ;
    private Achat[] historiqueAchats ;
    // etc...
    private String notesDuCommercial ; // peut être non renseigné!
}
```

Il convient de discuter de l'application des conventions de *Bean* :

- Le champ `référenceUnique` est immuable et caractéristique de l'objet `Client`. Il ne peut être modifié : on aura donc une méthode `getRéférenceUnique()` mais certainement pas une méthode `set`.
Remarque : le champ devrait être marqué `final`. (Il pourrait même être `public final` si on considère qu'il n'y a pas d'évolution possible pour les codes de gestion le concernant).
- On peut décider ou non que la même règle s'applique pour les champs `nom` et `prénom`. Il s'agit là d'une décision de gestion (peut-être que, finalement, on peut changer le nom du client!).

- Le champ adresse est modifiable : on aura donc un set et un get
- Le fait d'être modifiable ne garantit pas la présence d'une paire accesseur/mutateur. C'est le cas pour le champ pointsFidélité : on peut envisager un get mais mettre en place un set **casserait l'encapsulation!**

En effet ce champ est modifié par ajout d'un nombre de points. Selon les règles de fonctionnement cela pourrait se produire quand on enregistre un achat ou même quand un commercial décide d'offrir des points à son client (méthode ajoutPoints(int nombre)).

Offrir une méthode set reviendrait à permettre à d'autres codes d'intervenir dans les règles de gestion! De plus si un tel code "client" fait d'abord un get pour ensuite faire un set cela peut poser des problèmes de concurrence d'accès (entre les deux un autre code a fait une modification de la valeur).

Le champ historiqueAchats est le type même du champ pour lequel l'encapsulation est primordiale :

- Au cours de l'évolution du code il y a de bonnes chances pour que le type change (et évolue en ArrayList par exemple).
- On voit bien ici qu'un set n'aurait pas de sens.
- Même le get pose problème : si on fournit une référence vers ce tableau à des codes clients ceux-ci peuvent en modifier le contenu sans passer par les règles de gestion dont la classe est gardienne! On devra donc passer un clone du tableau en résultat de la méthode (et, dans une version ultérieure, invoquer toArray sur l'ArrayList).

Ces considérations sur les propriétés des champs trouvent leur prolongement naturel dans la conception des constructeurs.

Si les règles de gestion considèrent qu'un Client a obligatoirement une référence, un nom, un prénom et une adresse alors il y a un moins un constructeur "minimum" qui garantit la présence de ces champs. Il peut les obtenir par calcul (par ex. la référenceUnique), par des consultations de ressource, ou parce qu'on les lui passe en paramètre (nom, prénom, adresse).

On ne peut PAS avoir un constructeur sans paramètre qui permettrait de créer un objet "incomplet" par :

```
Client client = new Client() ; // EXCLUS!
```

On notera, par contre, que le champ notesDuCommercial peut avoir une valeur à null. On devra donc documenter cette propriété (dans la documentation du get en particulier) et les codes clients devront tester le résultat de l'appel du get.

On notera également qu'il est possible d'avoir une méthode comme:

```
public BigDecimal getMontantTotalDesAchats() ;
```

sans qu'il y ait de champ correspondant (on a ici un attribut calculé).

Formations JAVA sur ce sujet

Pour approfondir ces notions (encapsulation, contrôles de responsabilité, objets "valeur", etc.) vous pouvez suivre les cours suivants :

- [USL110 - Java : Bases du Langage](#) (pour programmeurs débutants)
- [USL275 - Java : Programmation pour Développeur Confirmé](#)
- [UJD110 - Programme pour Développeur Débutant](#) (même programme mais sur 7 jours)

Pour approfondir la notion de *Bean* et préciser les cas où un constructeur par défaut sans paramètre est nécessaire voir :

- [IIN43: mise à niveau et préparation à la certification programmeur](#)
- Nos formations "java avancé" adaptés à la demande.

Dans le cadre du développement de composants Java EE :

- [UFJ310: Développement d'Applications pour la plateforme Java EE](#)
- [USL351: Développement de Composants d'Entreprise Avancés avec la Technologie EJB](#)

Bernard AMADE

© Demos