SpaceFox

Zeste de savoir

Ne générez pas vos getters et setters !

8 décembre 2022

Table des matières

Introduction

Les concepts *getters* et *setters*, aussi connus sous le nom de «accesseurs» et «mutateurs» dans la langue d'Estelle Faye, partent à l'origine d'une bonne idée. Et comme beaucoup de bonnes idées, celle-ci a été dévoyée en un grand n'importe quoi qui pose beaucoup plus de problèmes qu'il n'en résous.

Java *inside*, mais pas que

Ce billet parle de problématiques qui touchent surtout le monde Java, mais la dernière section élargit le concept à d'autres langages objet.

1. Au commencement était une mauvaise question

Les accesseurs et mutateurs permettent d'isoler les attributs d'une classe du monde extérieur : au lieu d'utiliser des attributs *publiques* que n'importe qui peut modifier n'importe comment, on utilise des méthodes pour les lire et les définir, ce qui a plusieurs avantages :

1. On peut ajouter des contrôles pour éviter que ces attributs ne prennent des valeurs invalides.
2. On peut renvoyer autre chose que l'objet lui-même – par exemple une copie immuable, très pratique pour les collections.
3. On profite de tous les avantages associés aux méthodes, comme le polymorphisme, la possibilité d'implémenter une interface explicite ou l'héritage.
4. On évite d'exposer la représentation interne de l'objet.

Le problème survient quand on en arrive à utiliser ces outils pour répondre à cette question:

? La mauvaise question

Comment est-ce que j'accède aux champs (privés) de ma classe depuis l'extérieur ?

La solution ultra-classique consiste à doter ces champs, de façon un peu machinale, d'accesseurs et de mutateurs (c'est tellement facile avec les IDE ou bibliothèques modernes), et roule ma poule. Et donc on a des classes pleines de `getMachin()` et de `setBidule(Object bidule)` qui ne font ni le moindre contrôle, ni la moindre transformation, et on ne s'est pas rendu compte de l'énorme problème que pose la question à laquelle on a répondu :

D'un point de vue de l'encapsulation, elle n'a aucun sens. Pourquoi ?

Parce que, par définition, on *ne veut pas* accéder aux champs privés de la classe depuis l'extérieur !

Ces champs sont la représentation **interne** de la classe, ils n'ont pas à être exposés à l'extérieur. Jamais. Sous aucun prétexte.

Par contre, les méthodes publiques de votre classe forment un contrat d'interface – qui n'a pas besoin d'être explicitée sous la forme d'une `interface` séparée ; cette interface doit être conçue, réfléchie: quelles sont les méthodes que cette classe doit fournir pour être utilisable ?

Il se **peut** que l'implémentation de certaines méthodes se résume à renvoyer ou définir un champ interne de la classe. Mais ce cas doit être une **conséquence** (plus ou moins par coïncidence) des choix d'interfaces et de représentation interne, et pas une exposition de la représentation interne en soi. En particulier, on doit pouvoir modifier cette représentation interne sans toucher le contrat d'interface – et inversement (pensez au renommage d'un champ ou d'une méthode mal nommée).

Des accesseurs ou mutateurs qui exposent sans raison l'état interne d'une classe, ça n'est pas anodin. Ça apporte tout un tas de problèmes dont certains peuvent être particulièrement pénibles :

- Des problèmes de valeurs incohérentes si le mutateur ne dispose pas des bonnes protections, ou si elles sont impossibles à implémenter dans un simple mutateur ;
- Des problèmes de concurrence si un mutateur permet de modifier un champ pendant son traitement par une autre méthode ;
- Des problèmes de sécurité, le plus classique étant celui-ci: la classe contient une collection, diverses méthodes permettent de la manipuler avec tous les contrôles nécessaires… mais un mutateur oublié dans un coin permet tout simplement de *remplacer* l'intégralité de la collection (l'objet lui-même, pas son contenu), ce qui court-circuite toutes les protections de toutes les autres méthodes.

2. Le problème des « beans » (les structures de données idiotes)

Il existe tout de même un cas où ces accesseurs et mutateurs sont utiles, voire indispensables: celui des structures de données, ou «beans» dans le langage Java. C'est ces classes qui ne servent en réalité qu'à stocker des données, et à faire quelques contrôles dessus.

On *pourrait* utiliser des attributs de classe publics pour remplir ce rôle dans beaucoup de cas, mais – au moins en Java – les pratiques habituelles, ainsi que beaucoup d'outils, supposent que ce rôle va être rempli par une classe avec des champs privés et des accesseurs et mutateurs.

La règle d'or à suivre est celle-ci:

Une structure de données ne doit contenir aucun code métier d'aucune sorte.

La seule logique acceptable au sein de ces classes, c'est celle qui permet de garder la cohérence interne des données. On peut aussi y trouver des métadonnées qui serviront à la sérialisation ou désérialisation de la structure (en JSON par exemple).

La question des objets immuables

Quand on parle de structures de données ou de «beans» Java, vient la question de savoir s'ils doivent être immuables[1] – c'est-à-dire que l'objet ne peut pas être modifié une fois créé.
Il n'y a pas de vérité absolue à ce sujet: avoir des objets immuables procure énormément d'avantages, notamment ceux d'éviter tout un tas de problèmes de cohérence et de concurrence. Par contre, certains algorithmes provoquent la génération de *beaucoup* d'objets à la durée de vie très courte s'ils sont utilisés avec des objets immuables, ce qui introduit une forte pression sur le système de gestion mémoire[2].
Une règle générale qui fonctionne bien, c'est d'utiliser autant que possible des objets immuables, et de surveiller l'utilisation mémoire pour corriger les problèmes qui apparaîtraient en conditions réelles.

3. Bannissez la génération automatique d'accesseurs ou de mutateurs

La conséquence de tout ça, c'est que toute méthode qui ressemble à un accesseur ou à un mutateur (`getBidule()` ou `setMachin(Object machin)`) doit être réfléchie et faire partie du contrat d'interface de la classe ; ou bien n'exister que pour implémenter une structure de donnée «idiote», sans logique métier.

Dans tous les cas, bannissez la génération automatique d'accesseurs ou de mutateurs !

— Sur les classes avec du code métier, implémenter la méthode «à la main» vous permettra de réfléchir à si elle est réellement nécessaire au contrat d'interface ;
— Sur les structure de données, il existe des outils bien plus pratique que la fonction intégrée de votre IDE, qui permettent d'éviter de transformer votre code en une soupe illisible de méthodes générées: les `record` pour Java 17+[3], Lombok pour n'avoir qu'une poignée d'annotations au lieu de longues listes de méthodes sans intérêt, ou même… des champs publics pour les structures de données internes à une classe qui n'ont pas besoin des fonctionnalités apportées par les fonctions. Non, ça n'est pas sale.

1. Ou *«immutable»* par calque avec l'anglais et le mot «mutable».
2. Mais pas tellement plus de fuites mémoire, au contraire: beaucoup de fuites mémoires sont précisément dues à un objet mutable qui a conservé des instances qu'il n'aurait pas du. Les grosses fuites mémoire par des objets immuables sont souvent assez faciles à tracer et corriger.

4. «Mon langage est meilleur, il n'a pas d'accesseurs ni de mutateurs comme fonctions, mais un système intégré»

D'autre part, si votre outil d'analyse de code (au hasard, SonarQube dit «Sonar») râle sur le fait que vous n'avez pas d'accesseurs ou de mutateurs: sachez que ces outils se configurent très bien, et que les réglages par défaut ne sont qu'indicatifs. Parlez-en à votre administrateur (et s'il reste psychorigide sur ces règles… allez voir ailleurs, parce que c'est vraiment pas bon signe).

4. « Mon langage est meilleur, il n'a pas d'accesseurs ni de mutateurs comme fonctions, mais un système intégré »

Là je pense au moins à C# et à Kotlin – entre autres.

En fait, c'est exactement la même chose : ces fonctionnalités devraient être utilisées pour permettre de remplir de la façon la plus efficace un contrat d'interface, et pas pour exposer la représentation interne de la classe. La principale différence c'est que ces langages ont des *raccourcis d'écriture* qui permettent de faire comme si on manipulait directement un champ de la classe, alors que non.

Les règles sont donc les mêmes: tous les champs devraient être privés (et immuables) par défaut, les mutateurs et accesseurs exposés devraient se conformer à un contrat d'interface prédéterminé, et les structures de données utiliser les possibilités du langage et ne pas contenir de code métier (`data class` en Kotlin).

Conclusion

En résumé, l'utilisation systématique d'accesseurs et de mutateurs est un mauvais réflexe, un culte du cargo porté par de trop longues années de mauvaises habitudes et de d'incompréhension du modèle objet, en particulier de l'encapsulation.

Réfléchissez à la conception de vos classes, et n'utilisez la génération automatique d'accesseurs et de mutateurs que pour des structures de données dépourvues de toute logique métier.

Les personnes qui devront comprendre et maintenir votre code – y compris vous-même dans quelques mois – vous en remercieront.

---

3. En fait avant, mais les versions de Java antérieures à la 17 qui gèrent les `record` étaient des versions intermédiaires dont le support a expiré aujourd'hui. La première version *supportée* (à long terme) de Java qui permet de les utiliser est bien Java 17.