SpaceFox

Zeste de savoir

Un point sur la programmation objet (POO)

20 juin 2023

Table des matières

Introduction

Comme le sujet revient comme un marronnier, voici un billet sur la programmation orientée objet et pourquoi elle est mal utilisée et enseignée.

Ce billet repose principalement sur ces discussions du forum: [1] [2] . D'autre part, il présente une version pragmatique de la POO et repose surtout sur la pratique; il peut donc parfois s'éloigner des théories développées dans le cadre de l'étude formelle des langages de programmation.

Il ne représente que ma vision de la chose et n'engage que moi; vous êtes les bienvenus pour en discuter –avec respect et cordialité– en commentaires.

1. À propos de la programmation orientée objet en elle-même

1.1. Qu'est-ce que la programmation orientée objet ?

La programmation orientée objet – ou POO – est un paradigme de programmation informatique qui organise le code en un assemblage de briques appelées **objets** (merci *Captain Obvious*).

Ces objets ont un **état interne** et un **comportement**, qui leur permet d'interagir entre eux.

Et… c'est tout.

Si le principe d'encapsulation est correctement respecté (ce qui devrait toujours être le cas en théorie mais ne l'est pas en pratique), l'état interne d'un objet est inaccessible de l'extérieur, et ne peut pas être manipulé directement.

La conséquence immédiate de tout ça, c'est que la phase de conception est primordiale, parce que c'est la définition correcte des objets et de leurs interactions qui va permettre de faire fonctionner le programme.

1.2. Qu'est-ce que *n'est pas* la programmation orientée objet ?

Beaucoup d'éléments de langages de programmation sont souvent considérés comme «indispensables à la POO» alors qu'en fait pas du tout. Comme les classes ou les prototypes, le typage statique et/ou fort, les interfaces, etc.

La programmation orientée objet n'est pas non plus – et là je vais avoir tous les théoriciens du langage sur le dos – une application stricte des principes SOLID . C'est un simple problème de pragmatisme: ces règles sont simplement des indications qu'il est préférable de suivre, pas des lois absolues – et donc parfois il est plus simple de ne pas les suivre.

Cela dit, suivez SOLID autant que possible, ça ne fera pas de mal.

1.3. À quoi sert la programmation orientée objet ?

D'abord à regrouper le code en unités cohérentes, ce qui passe par une encapsulation correcte et une définition claire des interfaces.

Ensuite à permettre le découplage entre le fonctionnement interne d'un objet et son comportement: si on respecte les contrats de comportements, on doit pouvoir changer tout le fonctionnement interne de l'objet sans conséquence fonctionnelle.

1.4. À quoi *ne sert pas* la programmation orientée objet ?

La programmation orientée objet *permet* la réutilisation «simple» du code (via la réutilisation des objets), mais ça n'est pas du tout une caractéristique propre à ce paradigme. On réutilisait du code bien avant la POO.

Ça n'est pas non plus un paradigme orienté performances: le découplage et la cohérence du code (et donc sa maintenabilité) sont prioritaires. Ce point fait que la POO n'est pas très utilisée dans les domaines qui ont un besoin massif de performances, comme le jeu vidéo ou le calcul scientifique.

2. Et donc, c'est quoi le problème ?

Au fait, «programmation orientée objet» c'est très long, donc à partir de maintenant je vais surtout utiliser «POO». D'accord? Oui?Merci!

2.1. « Quand tu as un marteau, tout ressemble à un clou »

La POOdevrait être utilisée (et devrait n'être utilisée que) lorsqu'on a un problème qui peut se traduire en objets qui interagissent entre eux.

Mais comme dit le proverbe, «Quand tu as un marteau, tout ressemble à un clou», et donc la tentation est grande de créer des objets très artificiels, qui n'ont à peu près aucun sens, juste pour «faire de la POO» (quelle qu'en soit la raison à l'origine).

Alors qu'en fait il faudrait limiter l'usage de la POO aux parties du code pour lesquelles c'est pertinent, et utiliser d'autres paradigmes pour gérer le reste. Le problème, c'est que ça n'est pas toujours facile parce que…

2.2. Un enseignement complètement à la rue

L'immense majorité des cours de POO n'expliquent pas ce qu'est la POO: ils expliquent une *vision défectueuse* de la POO. Généralement en lui prêtant des buts et des moyens qui ne sont pas les siens, ce qui conduit à une mécompréhension générale de l'outil.

Le pire étant les exemples donnés, qui sont très souvent:

— Mauvais (entre autres parce qu'ils ne définissent généralement que l'héritage et pas les autres formes d'interaction comme la composition)
— Impossibles à mettre en pratique dans un vrai programme, ce qui n'aide pas à comprendre. Sérieusement, qui a besoin de représenter «un chat» ou «une voiture» sous cette forme dans un programme?
— Le simple fait de prétendre que tout problème peut être résolu par la POO. Je ne compte même plus les exercices où on exige de l'élève qu'il utilise la POO pour résoudre un problème où ça n'a aucun sens.

Par pitié, arrêtez de donner des exemples à base d'animaux ou de véhicules quand vous expliquez la POO! Ils sont totalement absurdes et se contentent d'embrouiller le monde! Même les exemples à base de figures géométriques sont douteux, parce qu'ils conduisent généralement à une violation du principe de substitution de Liskov – un cas qui sera discuté plus loin.

2.3. Des langages qui font n'importe quoi

On a un looong historique de langages «orientés POO» qui ont fait n'importe quoi avec. Soit comme PHP < 5.3 qui prétentait permettre d'en faire alors que non, soit comme Java (surtout < 8) qui forçait en pratique à en faire là où ça n'était pas utile.

Si je prends l'exemple de Java que je connais le mieux (vu que je bosse avec depuis près de 15 ans), on peut noter en vrac:

— Tous les types natifs qui ne sont pas des objets, mais qui ont des autoboxing / autounboxing vers des classes équivalentes pour pouvoir les considérer comme tels (avec tous les pièges et comportements bizarres que ça peut entrainer).
— L'obligation de déclarer une classe pour tout et n'importe quoi, y compris ce qui n'est de doute évidence pas orienté objet (coucou la fonction `main`[1]). Des objets planqués sous le tapis (dans les Enum par exemple).

— Des chaines d'héritage qui n'ont aucun sens et qui conduisent à des comportements aberrants[2].

Et là je parle bien du langage lui-même, et pas de l'utilisation qui en est faite.

Et puisqu'on en parle…

2.4. Le cas particulier de l'héritage

Le concept d'héritage est sans doute le plus polémique, mal compris et mal utilisé de la POO. Et pourtant il est utilisé en masse.

Le principal problème de l'héritage, dans la vie de tous les jours, c'est qu'il conduit très vite à un code incompréhensible et difficile à maintenir, où pour comprendre le **comportement** d'un objet *(et je rappelle que c'est l'un des principes de base de la POO!)* on doit remonter une pile complète d'héritage, en vérifiant à chaque étape si la méthode du comportement qui nous intéresse n'est pas redéfinie.

Un système d'héritage mal conçu peut aussi engendrer des effets de bords indésirables, avec une modification interne d'un objet parent qui casse le comportement des objets fils.

Si l'héritage est mal utilisé, c'est à la fois par mimétisme (beaucoup de langages ou *frameworks* orientés objets ont des chaines d'héritage à rallonge dans leurs API), par mauvais enseignement et par mauvaise compréhension. De plus, l'héritage peut être *performant* par rapport à d'autres techniques comme la composition[3].

En fait, on a tendance, quand on utilise la POO, à utiliser l'héritage dès qu'on a du code –ou pire: des données– à factoriser (hop, le code commun part dans un objet parent) sans que ça soit pertinent. Il y a d'autres techniques, comme la composition ou les traits qui répondent souvent mieux au besoin[4].

Si l'héritage s'avère être une solution pertinent, la solution **devrait** respecter le principe de substitution de Liskov , dit «LSP» (c'est le `L` du principe SOLID mentionné plus haut). Et je dis bien «devrais» et pas «doit», parce que c'est une règle informelle[5] et pas une règle absolue. En particulier, elle part d'un principe qui s'avère assez faux en pratique dans les programmes réels: celui de la substituabilité . En pratique, l'usage des objets fils n'est pas totalement substituable aux objets parents et ne cherche pas à l'être, les objets fils sont plutôt utilisés comme «des objets parents avec des propriétés en plus», des objets parents qui sont *aussi* des objets fils. Le cas le plus commun de violation du LSP, c'est les objets parents abstraits, sans implémentation.

---

1. En instance de disparition avec la JEP 445 qui devrait arriver avec Java 21.
2. Par exemple, l'interface `List` expose des méthodes de modification, qui sont donc présentes y compris sur les listes non modifiables, qui ont donc des méthodes qui lancent des exceptions quand on les appelle.
3. Je n'ai plus les chiffres exacts sous la main, mais le temps d'accès à une méthode, même avec du polymorphisme, est presque constant dans les langages à JVM (Java, Kotlin…) quelle que soit la profondeur d'héritage, même déraisonnable (plusieurs centaines de niveaux). La même structure représentée avec de la composition a un temps d'accès grossièrement linéaire avec la profondeur de composition. Ça peut être différent avec d'autres langages, à vérifier.
4. L'héritage *privé* peut être aussi une bonne solution, sous certaines conditions – la première est que la possibilité existe dans le langage utilisé…
5. D'après Barbara Liskov elle-même .

En résumé: l'héritage ne devrait être utilisé surtout pour *modifier* des comportements. On peut s'en servir pour *ajouter* des comportements, à condition d'être à l'aise avec le fait que ça va provoquer des choses bizarres du point de vue du LSP.

Cela dit, comme le principe SOLID en général: vous avez tout à gagner à respecter *au maximum* le LSP.

2.5. Des pratiques habituelles complètement cassées

Tout ça conduit à des pratiques habituelles considérées comme «de la POO» (voire imposées en entreprise) qui n'ont en fait rien à voir avec la POO, et qui sont totalement contre-productives. En vrac:

— Cette manie de mettre des getters et setters partout . Ça expose l'état interne de l'objet et casse complètement la notion d'encapsulation donc d'état interne (donc non exposé à l'extérieur).
— Tout faire par héritage. Il y a des cas où ça n'a juste pas de sens, la composition c'est bien aussi.
— Les piles de `instanceof` pour déterminer le comportement à appliquer en fonction du type de l'objet. Ça déporte la logique dans le code appelant, au lieu de laisser les objets réagir correctement à leur manipulation. Accessoirement, c'est un signe que le LSP est cassé sur l'objet en question.
— Le classique «il doit impérativement y avoir des interfaces » (et ses petits frères, les Factory etc). Là c'est plus qu'on se rajoute du travail inutile sous de faux prétextes.

3. Mais alors, qu'est-ce qu'on fait ?

L'idée, c'est d'utiliser un outil adapté à chaque problème, et donc plusieurs paradigmes au sein d'un même programme. La plupart des langages modernes sont d'ailleurs multiparadigmes et assez souples là-dessus. Idem avec les évolutions modernes de langages anciens, comme Java qui a ajouté une bonne dose de fonctionnel depuis Java 8 – et dans les cas où c'est utile, comme les traitements de séries d'objets, c'est très appréciable.

Il y a trois inconvénients à ça:

1. Il faut maitriser plusieurs paradigmes différents en même temps et être capable de changer facilement selon le contexte, ce qui n'est pas facile.
2. On voit arriver **de nouveaux** problèmes. Comme cette mode qui veut que l'on ne crée plus que des objets immuables, ce qui est très pratique pour les traitements fonctionnels et peut sensiblement améliorer les performances… mais qui est contraire à la définition d'«objet» dans la POO. Et donc appeler ça des «objets» devient abusif, casse les parties de code où ils sont utilisés dans le paradigme de POO, et engendre encore plus de confusion[1].
3. Il faut être capable d'identifier le paradigme le plus efficace en fonction du contexte. Et ce même quand ce contexte est mensonger, comme quand un langage se déclare «orienté objet» mais permet de faire aussi du procédural, du fonctionnel, de l'impératif[2] et j'en passe… ou quand un exercice parle beaucoup de POO alors que c'est sans doute l'un des pires paradigmes pour le réaliser .

D'autre part, lorsque vous utilisez la programmation orientée objet (parce que c'est une solution adaptée au problème en cours, donc), revenez toujours aux fondamentaux. Pensez en termes d'**état interne** et de **comportement**, et fuyez tout autre vocabulaire (en particulier: *champ*, *attribut*, *propriété* qui sont de bons candidats pour conduire à une conception cassée).

Conclusion

Voilà, c'était ma vision de la programmation orientée objet à à fin juin 2023. J'aurais pu rajouter plein de détails mais ce billet est déjà beaucoup trop long.

Merci à @lmghs, @pierre_24, @gbdivers, @entwanne et tous les gens que j'oublie pour leurs réflexions pertinentes qui ont nourri la mienne.

Je vous laisse pinailler dans les commentaires!

1. Notez que l'auteur de ces lignes est néanmoins partisan de l'usage d'«objets immuables» dès que possible. Parce que c'est quand même vachement pratique!
2. Je pars ici du principe que vous utilisez ici un langage orienté objet *mainstream*, qui sont tous procéduraux et impératifs. La situation pourrait être différente avec des langages exotiques.