ARTRAGIS

Zeste de savoir

[Réaction]Programmation orientée objet, qu'est-ce que c'est

24 avril 2021

Table des matières

Aujourd'hui, je vais tester quelque chose : réagir à un article écrit par des journalistes sur un sujet technique.

Sur discord , des gens ont un peu ricané lorsque la section "Futura Tech" du site futura-sciences a publié un article nommé "Programmation orientée objet : qu'est-ce que c'est" .

La plupart de nos réactions étaient négatives, mais je me suis dit qu'on pouvait quand même tirer quelque chose de tout ça, notamment un débat sur ce que c'est ou ce que n'est pas la POO.

!

Ma réaction va être biaisée, mais je peux déjà vous dire qu'elle le sera à cause de quelque chose de très précis: mon propre apprentissage de la POO a été marqué par deux événements. D'un côté, un cours particulièrement chiant, mais où l'enseignant a réussi à montrer l'importance capitale du *principe de responsabilité unique* et pourquoi la *composition* était plus importante que l'héritage, de l'autre une conférence d'un des créateurs de Small Talk qui encensait JavaScript pour être (de mémoire) "le langage à la POO la plus pure".
Vous avez le décors.

1. Définition de la POO

Commençons par les présentation:

> La programmation orientée objet est une méthode de programmation informatique de plus en plus plébiscitée, que ce soit dans le développement logiciel ou la data science.

Ah. C'est vrai ou pas au moins ça ? Depuis le temps qu'on nous dit que la POO est dépassée, je me demande si vraiment elle est de plus en plus utilisée ?

Surtout quand je vois à quoi un playbook jupyter de data-science ressemble, je me dis qu'en fait cette affirmation est un peu désuette.

D'ailleurs, on y reviendra plus tard mais un des langages qui fait de la concurrence à python dans les data science c'est go, @nohar vous donnera sûrement un avis plus éclairé que le mien, mais en fonction de comment on définit "La POO", on ne répondra pas toujours la même manière à la question "Go est il un langage objet".

D'ailleurs, c'est probablement là qu'est le soucis quand on présente la POO et comment la faire: vu le nom on se doute que la POO est une manière de programmer avec des objets. Mais c'est quoi un objet ?

1.1. Un objet

L'article a une réponse à vous apporter:

> On appelle objet, un ensemble de variables complexes et de fonctions, comme par exemple un bouton ou une fenêtre sur l'ordinateur, des personnes (avec les noms, adresse…), une musique, une voiture…
> Presque tout peut être considéré comme un objet.

Si avec ça vous avez compris, bravo à vous.

Outre la construction de la phrase, elle ne vous dit clairement pas ce qu'est un objet *dans le domaine de l'informatique*. Parce que bon, étaler les synonymes de objets (babioles, trucs, bidules…) ça ne vous apprend rien.

Heureusement, l'auteur a pensé à vous et va vous donner la clef de compréhension:

> La première [étape de la POO] consiste à modéliser les données en identifiant les objets que le programmeur souhaite manipuler ainsi que leurs interactions.

Vous savez toujours pas ce que sont les objets, mais vous savez qu'ils *interagissent* et c'est c'est beau.

Je me moque, mais si on se rappelle de mon disclaimer de début d'article, c'est en effet primordiale. Je vais donc vous donner une **première** définition **personnelle** de la POO.

Première définition personnelle

La programmation orientée objet consiste à extraire du besoin fonctionnel les intéractions entre les différentes entités prenant partie au programme: une interface avec l'humain, une source de données, un algorithme à exécuter. Une fois ces intéractions extraites on les modélises dans des *briques logicielles* ayant une responsabilité unique et *déléguant* le reste de l'algorithmes aux entités qui l'entourent.

C'est encore assez vague, mais après avoir écrit cette phrase, j'ai décidé d'aller sur wikipédia, et j'ai vu ça:

> La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique. Elle consiste en la définition et l'interaction de briques logicielles appelées objets; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.

> Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

On remarquera que les exemples évoqués par futura ressemblent à la partie "entité du monde physique" de cette définition.

D'ailleurs je me dis que le plus gros problème de la définition de futura c'est qu'elle fait l'erreur d'isoler "objet" pour tenter de retomber sur ses pieds. Mais finalement que ça soit du côté de wikipédia ou de ma définition, ce qui importe c'est qu'on programme et qu'on va juste **concevoir** notre programme pour mettre en avant des intéractions. C'est pour ça que des choses très *abstraites* apparaissent.

Je pense que la plus gros indice qui va dans le sens de cette analyse c'est cette phrase:

> Une fois l'objet créé, il communique avec une interface bien définie appelée message.

J'ai reproduit le texte et la mise en forme pour montrer combien l'article se méprend: il confond interface homme/machine, interface qui définit les intéractions entre objets (et souvent déclaré avec le mot clef `interface` tout est lié) et *message* qui a tellement de sens en informatique que je me demande vraiment si on est toujours dans le sujet de la POO.

1.2. Encapsulation, et pensée OO magique

(pensée oO: vous l'avez ?)

Si vous n'avez toujours pas compris ce qu'était la POO, ne vous inquiétez pas, peut être que finalement ce qui est intéressant, c'est à quoi ça sert.

Avant de passer à la réponse (qui est le seul truc choquant de l'article en fait, le reste n'est qu'approximations qu'on peut comprendre quand on essaie de présenter des choses ultra techniques à des non techniques) lunaire de futura-sciences, je vous propose la mienne:

- Tester les programmes plus facilement ;
- Travailler plus facilement en équipe, car tout en restant technique on peut facilement s'exprimer dans un langage naturel pour transmettre la connaissance ;
- Réutiliser des blocs de codes plus larges entre les projets qu'avec le paradigme dont la POO hérite: le procédural.

Les commentaires du billets sont ouverts pour donner d'autres intérêt si vous en avez ou nuancer ceux que j'ai proposés.

Trêve de teasing, qu'en dit futura ?

> En masquant les fonctions et variables de l'objet, le programmeur crée un code source parfois complexe mais facilement utilisable, tout en renforcant la sécurité du système et en évitant la corruption accidentelle de données. Grâce à ses propriétés d'héritage et de polymorphisme, une classe est réutilisable par le programme pour lequel elle a été créée, mais aussi par d'autres programmes orientés objet.

- La complexité d'un programme ne dépend casi pas du paradigme.
- La réutilisabilité est *selon moi* en effet augmentée, mais encore faut-il savoir *réutilisable par qui ?*.
- LA POO NE PERMET PAS D'AUGMENTER LA SECURITÉ NI LA ROBUSTESSE FACE AUX CORRUPTIONS, MERCI.
- Alors la réutilisabilité d'une classe n'est que peu corrélée au polymorphisme selon moi (ou alors il faut se faire des noeuds aux cerveau pour lier le polymorphisme à la réutilisabilité).
- Si votre langage le permet une fonction isolée est tout aussi réutilisable qu'une classe, même dans des autres programmes.

Outre la grosse erreur sur la sécurité, en fait ces "intérêt" sont globalement non pas les intérêts de la POO mais les intérêt d'un programme conçu pour être développé par plusieurs personnes sur le temps long. Il va donc falloir créer des bouts de codes réutilisables partout, des moyens de communiquer, des tests… La POO permet d'avoir un outillage pour ça, mais le procédural ou le fonctionnel aussi.

2. Les concepts clés

Avant de partir, Futura nous donne une liste de six concepts clés de la POO :

- La classe
- Les objets
- L'encapsulation
- L'abstraction
- L'héritage
- Le polymorphisme

C'est là que la conférence "le JS est la forme la plus pure de POO" a son impact sur moi. A l'époque de cette conférence le JS possédait en effet un module orienté objet mais la notion de classe n'existait pas.

Si on se fie à l'article cela signifie que le JS ne pourrait donc pas profiter de l'encapsulation, de l'abstraction (très mal définie dans l'article, au demeurant), d'héritage ou de polymorphisme.

Alors qu'en fait… à cette époque JS avait tous les concepts sauf… les classes.

Car finalement ce qui compte, je pense, ce ne sont pas ces concepts clés là mais:

- Définition des modes d'intéraction: chaque "objet" définit la manière dont il peut être manipulé par les autres. Cette "manière d'être manipulée" est une simple liste d'ordres qu'on peut transmettre à l'instance d'un objet. Tous les objets qui sont capables de répondre à une liste d'ordres données peuvent être utilisés tels quels dans les différents algorithmes.
- Notion d'instance : Une instance c'est un objet avec un état donné. C'est lui qui "répondra aux ordres".
- Délégation et composition : Chaque objet n'est capable de faire qu'exécuter ses ordres à lui, mais peut dépendre d'autres objets pour fonctionner.

3. Un exemple ?

Ce billet commence déjà à s'éterniser alors que l'article de futura science était très court. Pourtant j'ai envie de vous donner un exemple fictif de POO qui permettra de montrer ma vision de la chose.

3.1. Énoncé

Notre but est de simuler un monde où des gens vont acheter des objets à des marchands.

Pour des raisons de sécurités évidentes, dans ce monde où la corruption et la délinquance font des ravages, il a été décidé que le paiement ne se ferait jamais face à face mais avec un programme (le notre donc).

Le programme doit donc être capable de prendre le paiement de l'acheteur et de l'acheminer jusqu'au vendeur.

Cependant, il est à noter que le vendeur ne reçoit pas l'argent directement, il le met en banque. De même que l'acheteur utilise des outils tels qu'une carte de paiement ou un chèque (oui ça existe encore) pour demander à la banque de payer.

Ah et dans ce monde idyllique, tout le monde n'utilise pas la même monnaie, mais l'État dans son infini sagesse nous donne le taux de change entre deux monnaies grâce à un composant logiciel qu'il a développé.

3.2. Ma conception

Il n'existe pas de "bonne réponse" à ce système, si ce n'est que comme c'est moi qui écrit le billet, je connais tous les sous-entendus de mon énoncés et donc que je me suis soit très simplifié le travail soit compliqué la tâche uniquement dans un but dialectique. Pour autant voir vos solutions m'intéresse.

pour ma part je vois notre système de paiement comme un immense tuyau où des gens capables de payer donnent de l'argent virtuel ou non à des gens capables de recevoir de l'argent.

1. acheteur "donne" de l'argent "virtuel" à sa banque (ordre par carte bancaire/chèque)
2. banque "donne" de l'argent à la banque de vendeur (argent réel cette fois-ci) et "prend" l'argent de l'acheteur (ligne dans le relevé de compte)
3. banque "donne de l'argent" "virtuel" au vendeur (impression d'une ligne dans le relevé de compte)
4. vendeur "prend" de l'argent à la banque lorsqu'il fait un retrait

Entre chaque étape de notre "tuyau", il y a quelque chose qui vient "convertir" les montants dans l'a monnaie du receveur.

Je vais donc créer quatre "ensembles d'ordres":

```
// on va supposer que Devise et SommeEnDevise sont des choses qui sont natifs au système car fournies par l'état)

interface Payeur {
    // false si paiement impossible
    boolean payer (Receveur receveur, SommeEnDevise argent)
    validerPaiement(Receveur receveur, SommeEnDevise argent)
    invaliderPaiement(Receveur receveur, SommeEnDevise argent)
    PayeurReceveur intermédiaireDePaiement()
}
interface Convertisseur {
    SommeEnDevise echanger(SommeEnDevise sommeAEchanger, Devise deviseDésirée)
}
interface Receveur {

   recevoir (SommeEnDevise argent, Payeur source)
   validerPaiement(Receveur receveur, SommeEnDevise argent)
       invaliderPaiement(Receveur receveur, SommeEnDevise argent)
   PayeurReceveur intermédiaireDePaiement()
   Devise getDevise()
}

interface OperateurPaiement {
   opérerLaTransaction(Payeur payeur, Receveur receveur,
       SommeEnDevise argentDepenséParAcheteur)
}
```

Pour moi, notre ”tuyau” est un opérateur de paiement en soit: il permet à la transaction entre l'acheteur (qui est un payeur) et le vendeur (qui est un receveur).

On notera que les banques sont à la fois Payeur ET Receveur.

Le système de paiement interbanquaire est aussi un opérateur en lui-même: contrairement au système global, il n'essaiera pas de trouver l'intermédiaire de la banque.

On pourrait globalement voir ces deux systèmes ainsi:

```
class SystemInterbanquaire implements OperateurPaiement {
  public void opérerLaTransaction(Payeur payeur, Receveur receveur,
      SommeEnDevise argentDepenséParAcheteur) {       Convertisseur convertisseur = Etat.getConvertisseur();
      // pas de frais de change entre banque, pas de ça entre nous :p
      payeur.payer(receveur, argentDepenséParAcheteur);
      receveur.recevoir(payeur,
          convertisseur.echanger(argentDepenséParAcheteur,
          receveur.getDevise());
```

```
6         payeur.validerPaiement(receveur, argentDepenséParAcheteur)
7         receveur.validerPaiement(payeur,
              convertisseur.echanger(argentDepenséParAcheteur,
              receveur.getDevise())
8     }
9   }
10
11  class SystèmeGlobal implements OperateurPaiement {
12
13   public void opérerLaTransaction(Payeur payeur, Receveur receveur,
          SommeEnDevise argentDepenséParAcheteur) {
14       // on peut en faire un attribut de classe
15       // ou bien utiliser le pattern factory
16       OperateurPaiement systemInterbanquaire = new
              SystemInterbanquaire();
17       // 42 c'est pour les frais de change entre banque, faut vivre
              ma petite dame
18       if (payeur.payer(payeur.getIntermediaire(),
              argentDepenséParAcheteur + 42)) {
19          payeur.validerPaiement(receveur, argentDepenséParAcheteur +
                42);
20          payeur.getIntermédiaire().recevoir(payeur,
                argentDepenséParAcheteur + 42);
21
                systemInterbanquaire.opérerLaTransaction(payeur.getIntermédiaire()
                receveur.getIntermédiaire(),
                argentDepenséParAcheteur();
22          receveur.validerPaiement(payeur,
                argentDepenséParAcheteur());
23        } else {
24           // on invalide
25        }
26   }
```

L'exemple n'est pas parfait, et la solution trouvée demande un peu de rafinement, mais elle permet vraiment de mettre en avant les notions que j'ai proposées tout à l'heure, je pense.

Elle montre aussi ce que c'est vraiment que le polymorphisme. Notamment elle montre que peu importe que l'auteur paie par chèque ou par carte, on peut faire le même algorithme à chaque fois. La "validation" du chèque étant certes différente de la "validation" de la carte bleue, elle n'en reste pas moins une validation du point de vue du système de paiement.