8

Les grands principes de la POO

Si vous demandez à des développeurs (surtout aux plus anciens) ce qu'est la programmation orientée objet (POO) et quelle en est l'idée générale, vous obtiendrez souvent une réponse proche de

> *Ben tu vois, la programmation orientée se base sur l'encapsulation...*
> *[La suite dépendra de la personne à qui vous vous adressez.]*

Les gens qui vous font ce genre de réponse sont, classiquement, des gens qui ont, pour leur malheur, appris la programmation orientée objet au travers d'une approche historiquement mal comprise.

Bien sûr, la programmation orientée objet tend à faire la part belle à l'encapsulation et donne tout ce qu'il faut pour en rendre l'usage et la mise en œuvre les plus naturels possible, ce qui en fait un principe majeur du paradigme orienté objet, mais ce n'est malgré tout pas la grande nouveauté de ce paradigme. Après tout, l'abstraction, qui est une forme d'encapsulation des données, était déjà possible en C : pensez à la structure FILE dont personne ne sait ce qu'elle contient, et personne n'a besoin de le savoir grâce aux fonctions qui la manipulent.

La grande nouveauté proposée par le paradigme orienté objet est ce que l'on appelle la *substituabilité* ou, si vous préférez, la capacité que l'on peut avoir à faire passer une variable d'un type particulier comme étant d'un autre type plus générique.

Au final, l'idée de base de la programmation orientée objet est de réfléchir aux différents types que nous allons utiliser en termes de *comportements*, de *services attendus/rendus* par ces types au lieu de réfléchir à ces types en termes de *données manipulées*.

Note *› La différence entre un service et un comportement est relativement mince, dans le sens où tous les services peuvent être considérés comme des comportements. Seulement, un service est un comportement qui peut être appelé* depuis l'extérieur, *alors qu'un comportement peut n'être qu'à usage interne.*

Quelle différence cela fait-il, me demanderez-vous sans doute ? Eh bien, tout simplement que l'on ne va plus considérer un type `Personne` (par exemple) comme étant une structure composée de son nom, de son prénom et des autres informations qui nous seront nécessaires, mais comme un objet qui devra être en mesure de répondre à certaines

questions comme "Quel est ton nom ?", "Quel est ton prénom ?", et autres questions similaires, voire qui pourra réagir à certains ordre comme "Déménage à telle adresse" ou "Marie-toi avec x ou y".

C'est sans doute pour cela que l'approche historique de la POO fait la part si belle à l'encapsulation car le fait de considérer les différents types sous l'angle des services et comportements qu'ils doivent être en mesure de remplir tend à transformer les données qui permettent de remplir les dits comportements en "détails d'implémentation".

Évidemment, pour qu'une conception puisse être efficace, certaines règles doivent impérativement être respectées. Bon, allez, n'oublions pas le fameux jeu de mots :

- Quelle est la différence entre la théorie et la pratique ?
- En théorie, aucune, en pratique, il y en a une.

et gardons à l'esprit qu'il y a parfois un écart entre la théorie et la pratique.

Je vais donc plutôt reformuler ma phrase sous la forme de "Pour qu'une conception puisse être efficace, certaines règles doivent être respectées, à moins d'avoir de bonnes raisons de ne pas le faire".

Ces principes ne sont, par chance, pas excessivement nombreux et se limitent, pour les principaux, à la loi de Déméter et à l'acronyme SOLID.

1. La loi de Déméter

Il existe une loi dite de Déméter qui, appliquée à l'informatique s'énonce à peu près dans les termes suivants :

> Si un objet de type A manipule en interne un objet de type B, l'utilisateur de A ne devrait pas avoir besoin de connaître B pour manipuler son A.

Un exemple pour bien comprendre

Mettons que nous voulions conceptualiser l'idée d'une voiture.

Il ne faudra pas réfléchir très longtemps pour se rendre compte qu'une voiture a besoin d'un réservoir à carburant pour pouvoir fonctionner, ni que ce réservoir donnera accès à un certain nombre de valeurs comme sa capacité maximale et la quantité de carbu-

rant dont il est actuellement rempli. Cependant, monsieur tout le monde n'aura *jamais* à manipuler directement son réservoir et même le garagiste ne devra y accéder que très rarement, voire jamais pour la plupart des voitures, afin de le remplacer par exemple. Les manipulations habituelles du réservoir passeront donc toute par l' *interface* que la voiture expose à l'utilisateur de son réservoir, à savoir :

- la trappe de remplissage ;
- la jauge du niveau de carburant ;
- d'éventuels indicateurs permettant d'évaluer la distance que l'on peut parcourir ;
- etc.

La classe `Voiture` va donc fournir une série de services relatifs à la présence du réservoir, ainsi que sans doute certains comportements qui y sont associés, mais n'a *a priori* strictement aucune raison de fournir un accès *direct* au réservoir dont elle est composée.

Pourquoi se passer des accesseurs ?

Vous aurez sans doute compris que, vu sous cet angle, il n'y a vraiment aucun intérêt à fournir un accesseur (*getter* en anglais) sur le réservoir, car cela ne correspond tout simplement pas à un service que l'on attend de la part d'une voiture : le réservoir n'est qu'une donnée qu'elle manipule pour accomplir certains services que l'on attend d'elle.

Lorsque vous envisagez de fournir un accesseur sur l'un des membres de vos classes, posez-vous donc d'abord la question "Est-ce que cela correspond à un service que je suis en droit d'attendre de sa part ?". Vous constaterez régulièrement que la réponse est, tout simplement, non.

Il peut y avoir des exceptions : si vous avez un objet qui doit être positionnable, il y a de fortes chances que l'un des services que vous en attendiez soit... le fait de fournir sa position et que ce service ne ferait en réalité que renvoyer la donnée en question. Mais il faut malgré tout considérer cela comme exceptionnel, dans le sens où vous devriez partir du principe que vous n'avez simplement pas à exposer des détails d'implémentation.

Par contre, il n'est pas du tout impossible qu'un service apporté par une classe particulière ne serve en réalité que de "passerelle" pour appeler le service correspondant d'un de ses membres.

Mettons que notre projet ait besoin de positionner différents objets dans un référentiel à deux dimensions. Nous allons sans doute créer une classe `Coordonnee` qui regroupe simplement une abscisse et une ordonnée afin de pouvoir représenter une coordonnée

particulière dans notre référentiel. Évidemment dans une telle classe, tant l'abscisse que l'ordonnée correspondent aussi bien à la donnée manipulée en interne qu'à un service que l'on est en droit d'attendre de la part de la classe car la classe doit être en mesure de répondre aux questions "Quelle est ton abscisse ?" et "Quelle est ton ordonnée ?".

L'une des implémentations possibles d'une telle classe (car le débat sur le sujet n'est pas clos) pourrait donc être très proche de

```
class Coordonnee{
    public :
       Coordonnee(int x, int y):x_(x),y_(y){}
       /* permet de récupérer l'abscisse de la coordonnée courante
        *
        */
       int x() const{return x_ ;}
       /* permet de récupérer l'ordonnée de la coordonnée courante
        *
        */
       int y() const{return y_ ;}
    private :
       int x_ ;
       int y_ ;
};
```

Il semble clair qu'une classe `Positionnable` dans notre projet fictif manipulera très certainement un membre de type `Coordonnee` afin de fournir les services que l'on attend de sa part. Si l'on s'en tient au pied de la lettre de la loi de Déméter, l'utilisateur de la classe `Positionnable` ne devrait même pas avoir connaissance de la classe `Coordonnee` pour utiliser la classe `Positionnable`. C'est d'autant plus vrai que nous aurons sans doute beaucoup plus souvent besoin de ne récupérer que la valeur de l'abscisse ou celle de l'ordonnée à laquelle se trouve notre objet `Postionnable`. Nous pourrions donc parfaitement nous contenter d'implémenter cette classe sous la forme de

```
Class Positionnable{
    public :
        /* permet d'obtenir l'abscisse à laquelle se trouve l'objet
         * courant
         */
        int x() const{return coord_.x() ;}
        /* permet d'obtenir l'ordonnée à laquelle se trouve l'objet
         * courant
         */
        int y() const{return coord_.y() ;}
    private :
        Coordonnee coord_ ;
};
```

Par contre, la question "Peut-on considérer que renvoyer la coordonnée de l'objet courant est un service que l'on est en droit d'attendre de celui-ci ?" reste ouverte.

D'un côté, on pourrait parfaitement estimer que, vu que l'on peut déjà récupérer l'abscisse et l'ordonnée, on n'a absolument aucune raison de vouloir être en mesure de récupérer la coordonnée dans son ensemble, et c'est d'autant plus vrai qu'il n'y a strictement rien qui oblige notre objet `Positionnable` à utiliser effectivement un objet de type `Coordonnee` comme membre : après tout, la classe `Coordonnee` est suffisamment simple pour que le concepteur du type `Positionnable` puisse encore choisir d'autres représentations internes de l'abscisse et de l'ordonnée.

D'un autre côté, si l'on a régulièrement besoin d'être en mesure de récupérer la coordonnée complète (comprenez : l'abscisse ET l'ordonnée), on pourrait parfaitement considérer que la classe `Coordonnee` est en quelque sorte un des "types primitifs" de notre domaine particulier et que l'un des services que l'on est en droit d'attendre de notre classe `Positionnable` est d'être en mesure de récupérer la coordonnée complète.

À vrai dire, les deux approches semblent tout à fait valables et montrent, si besoin en était encore, qu'il faut parfois accepter certaines entorses à la théorie.

Les mutateurs, c'est encore pire

Vous l'aurez sans doute compris par vous-même, si l'on n'arrive déjà pas à trouver une raison cohérente pour fournir un accesseur sur un détail d'implémentation propre à une classe, il n'y a strictement aucune raison pour fournir un mutateur sur cette chose.

Et même si un accesseur sur un détail d'implémentation existe, je vois malgré tout plusieurs raisons pour ne pas rajouter, en plus, un mutateur :

- Si c'est pour finir par donner un accès "plein et entier" au détail d'implémentation d'une classe, pourquoi devrions-nous perdre notre temps à placer ce détail dans une visibilité privée et à fournir un couple d'accesseur et de mutateur, alors qu'il serait si facile de le laisser en accessibilité publique, ce qui reviendrait tout à fait au même, le travail en moins ?
- Si vous permettez à l'utilisateur de votre classe d'accéder directement à un détail d'implémentation de celle-ci, non seulement il risque de multiplier les endroits dans le code où il profitera de cette opportunité, avec tous les problèmes que cela peut engendrer, mais en plus vous lui donnez une responsabilité qu'il ne devrait pas avoir : celle de s'assurer que les valeurs qu'il essaye de définir respectent l'ensemble des prescriptions que la classe impose à ces dernières. Avec pour conséquence un risque accru de sa part d'oublier l'une ou l'autre de ces prescriptions.

- Le nom de la fonction utilisé pour le mutateur (traditionnellement `setXXX` ou `setYYY`) est, par nature, beaucoup moins précis qu'une fonction qui modifie l'état interne de la classe comme par exemple `move(distanceX, distanceY)` ou encore `moveTo(newPosition)`.
- Le mutateur présente par nature une notion *de modification immédiate* : lorsque vous faite un `setPosition(newPosition)`, vous vous attendez à ce que, à l'instant T, votre objet se trouve à sa position de départ et à ce que à l'instant T+1, il se trouve à sa nouvelle position. En utilisant des fonctions comme `move(distanceX, distanceY)` ou encore `moveTo(newPosition)`, vous indiquez clairement que c'est un déplacement, et donc qu'il peut nécessiter *un certain temps*. Et, mieux que tout, vous laissez peut-être la possibilité à votre objet de ne pas atteindre sa position finale.
- Il devient particulièrement difficile de faire respecter des contraintes et des invariants multi-attributs : si vous développez une classe qui manipule en interne les valeurs x, y et z et que votre classe doit respecter en tout temps la contrainte que $x^2 + y^2 = z^2$, vous aurez énormément de mal à réaliser correctement la vérification de cette contrainte en ayant recours aux mutateurs.

Blanchisserie ou laverie automatique ?

L'un des aspects principaux de la programmation orientée objet est de vous inciter à réfléchir en termes de services rendus par vos objets et non en termes de données utilisées.

À ce titre, nous pourrions faire la comparaison entre une blanchisserie qui vous propose le service de nettoyage de vos vêtements, par quelqu'un qui connaît son métier, qui sait exactement quel produit utiliser en quelle quantité et à quelle température laver votre linge et une laverie automatique qui se contente de mettre à votre disposition des "données" qui correspondent aux machines et aux produits, à charge pour vous de savoir comment utiliser l'un et l'autre pour laver votre linge.

D'un côté, il n'y aura jamais de problème, votre pull en cachemire vous sera rendu dans un état impeccable, parfaitement repassé et aussi doux qu'au premier jour alors que de l'autre vous risquez de récupérer un pull tout à fait informe et assez rugueux pour décaper votre grille de barbecue des graisses brûlées qui se sont collées dessus lors de sa dernière utilisation.

Lorsque vous décidez d'exposer les mutateurs sur les différentes données manipulées par votre classe, vous décidez de créer une laverie automatique au lieu d'une blan-

chisserie. Et à partir de ce moment-là, tout pourra arriver car vous n'aurez plus aucun contrôle sur la manière dont votre classe sera utilisée.

2. Cinq piliers, c'est du SOLID

SOLID est l'acronyme qui regroupe cinq des principes pour lesquels il faut vraiment avoir une excellente raison avant de décider de ne pas les respecter.

Il correspond aux termes :

- S pour SRP (*Single Responsability Principle*) ;
- O pour OCP (*Open / Close Principle*) ;
- L pour LSP (*Liskov Substitution Principle*) ;
- I pour ISP (*Interface Segregation Principle*) ;
- D pour DIP (*Dependancy Inversion Principle*).

Plus vous respecterez ces principes, plus votre projet aura des chances d'être facile à maintenir et évolutif.

3. SRP : Le principe de la responsabilité unique

On ne le répétera jamais assez : chaque type, chaque fonction ne devrait jamais avoir qu'une et une seule responsabilité, et s'en charger correctement.

S'il est relativement simple de déterminer la responsabilité d'une fonction, il est par contre plus compliqué de déterminer celle d'une classe ou d'une structure. Et pourtant, si vous arrivez à déterminer une responsabilité unique pour chaque classe ou structure que vous aurez à créer dans vos projets, vous ferez un pas énorme en direction d'une réutilisabilité et d'une facilité de lecture accrue de votre code, tout comme vous le feriez en ce qui concerne les fonctions.

Imaginons que vous ayez besoin, dans un de vos projets, d'une fonction qui fasse (dans l'ordre) le café, la vaisselle et la lessive avant de sortir le chien. Vous pourriez parfaitement, car rien ne vous l'interdit au niveau du langage : créer une seule grande fonction monolithique qui contient l'ensemble des instructions nécessaires pour arriver à ce but.

Seulement, cette fonction a de grandes chances de s'étendre sur plusieurs centaines de lignes, ce qui n'est, en soi, déjà pas vraiment des plus faciles à lire. On considère en effet, pour des raisons purement historiques, qu'une fonction ne devrait pas nécessiter plus de cinquante lignes de code, même si cette limite est à prendre avec souplesse.

De plus, si par la suite vous deviez avoir besoin d'une fonction qui fait la même chose dans un ordre différent, vous en viendriez sans doute à écrire un code comportant exactement les mêmes instructions, mais dans un ordre différent.

Par contre, si vous déterminez directement une responsabilité unique (faire la vaisselle OU faire la lessive OU faire le café OU sortir le chien), et que vous veillez à créer une fonction pour chacune de ces responsabilités, non seulement les fonctions seront beaucoup plus courtes et simples à lire, mais en plus vous seriez très facilement en mesure de fournir une fonction permettant d'effectuer ces actions dans n'importe quel ordre.

Il en va strictement de même pour les classes et les structures. Une fois que vous avez déterminé une responsabilité unique pour chaque classe ou structure, vous pouvez déterminer très exactement les services que vous serez en droit d'en attendre, et vous pourrez dès lors utiliser n'importe quelle combinaison de différentes classes ou structures comme s'il s'agissait de briques de base pour un mur, en étant sûr de ne pas vous retrouver avec des fonctions n'ayant strictement aucun sens pour une classe ou structure plus complexe.

Notez enfin que le respect du principe de la responsabilité unique vous facilitera énormément la tâche lorsqu'il s'agira de respecter les autres principes de SOLID.

Méfiez-vous des termes trop vagues

Méfiez-vous comme de la peste des termes dont le sens est particulièrement vague, comme "gérer", "manager" et d'autres encore. Avez-vous, par exemple, déjà réfléchi à ce que représente le fait de définir la responsabilité d'un type ou d'une fonction comme étant "de gérer les objets de type XXX" ?

Cette simple expression englobe tout à la fois le fait :

- d'être en mesure de construire des objets sur demande ;
- de maintenir les objets en mémoire afin de pouvoir y accéder en cas de besoin ;
- de pouvoir manipuler les différents objets dont on dispose ;
- de pouvoir détruire les objets devenus inutiles ;
- et j'en oublie certainement.

Si le maintien des objets et leur destruction une fois qu'ils sont devenus inutiles a de fortes chances d'échoir à un seul et même type d'objet, la construction des éléments et le fait de manipuler les différents objets disponibles échoira en revanche très souvent à deux types d'objets totalement distincts.

Imaginons une classe qui représente "n'importe quel élément susceptible d'être affiché" que nous appellerons `DrawableItem` et sur laquelle je reviendrai un peu plus loin. Il est plus que possible que vous ayez une quantité phénoménale de types qui héritent, de manière directe ou indirecte, de cette classe de base, vu que de nombreux types utilisés dans votre partie métier devront pouvoir être affichés.

La gestion des différents éléments de types dérivés `DrawableItem` passera donc par le fait de les créer, d'en maintenir une liste en mémoire tant qu'on en a besoin, et forcément par la possibilité d'accéder aux différents éléments en fonction de différents critères afin de pouvoir les manipuler à notre guise. Sans oublier le fait de les détruire une fois qu'ils sont devenus inutiles

Le problème, c'est que si la création des différents éléments nécessite forcément de connaître tous les types dérivés de `DrawableItem` qui existent, tout le reste ne devrait à chaque fois n'envisager les différents éléments que comme des `DrawableItem` : il n'y a aucune raison cohérente de permettre à l'utilisateur de connaître le type réel des différents éléments une fois qu'ils ont été créés. Nous verrons d'ailleurs plus loin, à la section Le RTTI ne sert pas aux tests de types, que le fait d'accorder cette permission à l'utilisateur pose énormément de problèmes.

Or, si l'on en vient à considérer la gestion des éléments de types dérivés de `DrawableItem` dans son ensemble et à en déduire une seule classe qui prend tous les aspects que le terme "gérer" implique, le simple fait de devoir être en mesure de les créer nous oblige à exposer l'ensemble des types en question, avec comme conséquence un résultat beaucoup trop prévisible : l'utilisateur sera forcément tenté de recourir au RTTI de manière indue. Et cette tentation a généralement tendance à faire tâche d'huile au point d'envahir littéralement tout le module, lorsqu'elle se limite à un seul module.

Il est donc primordial de préciser exactement ce que l'on entend par le terme "gérer" (ou autres termes similaires comme "système", "manager", etc.) afin d'en déduire des responsabilités – et donc des concepts – beaucoup plus simples.

De cette manière, le maintien de la liste des différents éléments ira très certainement de pair avec la possibilité de détruire les éléments devenus inutiles et tout ce qui a trait à l'accès aux différents éléments, alors que la création des éléments et/ou leur manipulation réelle se fera au travers de concepts qui leur sont propres.