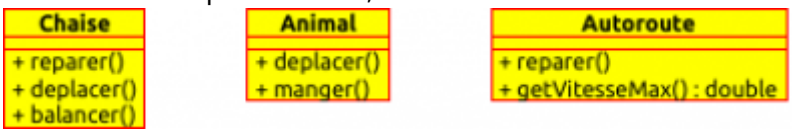
Bases de la conception Objet

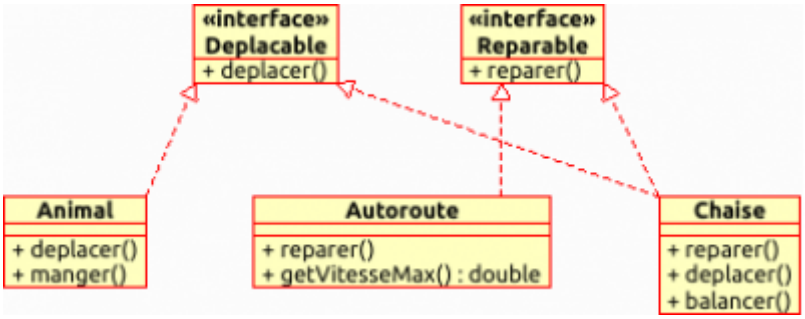
Les piliers de la POO

La POO s'appuie sur quatre piliers essentiels:

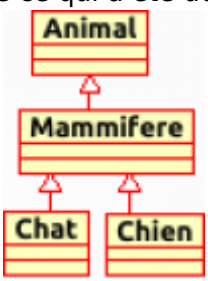
1. **l'abstraction**: un modèle est une abstraction de la réalité, une classe est donc un modèle (la classe Chaise n'est pas une chaise, mais un modèle de chaise). Cette abstraction peut être plus ou moins forte (déclaration d'un comportement attendu via une interface en java, classe abstraite, classe concrète). voir le sujet sur les modèles.
2. **le polymorphisme**: capacité à présenter plusieurs forme, concerne les méthode mais également les substitutions. Il y a quatre principales manières d'utiliser le polymorphisme:
    1. deux avec les méthodes exclusivement: la redéfinition (polymorphisme d'héritage) et la surcharge (polymorphisme paramétrique), lire https://web.maths.unsw.edu.au/~lafaye/CCM/poo/polymorp.htm;
    2. une plus générale avec le polymorphisme ad-hoc: adaptation d'un opérateur en fonction des types (le + entre deux entiers produit une addition des entiers, mais une concaténation entre deux chaînes de caractères), utilisation d'un nom de méthode identique pour une opération sémantiquement équivalente dans des classes différentes sans parenté ni lien fort conceptuellement;



    3. une dernière développant l'idée de nommage commun du polymorphisme ad-hoc qui consiste à garantir la substitution d'objets, classiquement en java en utilisant une interface pour déclarer un comportement et en faisant implémenter cette interface par les classes concernées:



3. **l'héritage** ou la **spécialisation**: possibilité d'exprimer le fait qu'un concept (une classe) **est un** autre concept (plus exactement **est une** spécialisation d'une abstraction de plus haut niveau) et donc de bénéficier (hériter) de ce qui a été définit dans le concept parent;



4. **l'encapsulation**: offrir la possibilité à un programme de n'exposer qu'une partie du code qu'il utilise en masquant le fonctionnement interne d'un objet. Une première manière simple de garantir l'encapsulation est de restreindre la visibilité des attributs en les déclarant **private** et

en utilisant des contrôles d'accès **lorsque cela est nécessaire** via les mutateurs (setters) et les accesseurs (getters); plus généralement, ***l'encapsulation ne se limite à la simple utilisation du mot-clef*** **private**, mais concerne tout ce qui contribue à masquer les fonctionnement interne et protéger les attributs de mutations par des tiers (par exemple, en clonant une référence dans un accesseur, ou en ne donnant pas les types concrets mais des super-types plus abstraits).

En résumé et guise de conclusion

La POO repose sur quatre piliers principaux que chaque langage orienté objet implémente avec ses particularités (exemple de la visibilité private ou protected en java ou en C++).

Ces piliers offrent des moyens de concevoir des logiciels souples, réutilisables, évolutifs et faciles à maintenir.

Cependant, gardez à l'esprit que ces piliers sont des outils: ce n'est pas parce qu'ils sont performants et permettent de faire des choses formidables que pour autant, vous avez la garantie de produire un programme de qualité. Ils ont besoin d'être manipulés correctement et à bon escient.

La section suivante explique comment mettre en œuvre une conception de qualité en se basant intelligemment sur ces quatre piliers.

Principes et bonnes pratiques de conception

Il existe de nombreux principes et bonnes pratiques pour la programmation objet.

Je présente ici ceux qui font référence aujourd'hui.

Ces principes s'appuient sur les quatre piliers présentés plus haut, et indique comment les utiliser correctement dans une conception objet.

Les principes SOLID ou SOIRS

SOLID est un acronyme proposé par Michael Feathers dans son ouvrage Working effectively with legacy code qui reprend 4 principes compilés par Robert Martin dans son article de 2000

Design Principles and Design Patterns

et un cinquième de 2002 extrait de Agile Software Development, Principles, Patterns, and Practices (au passage le site de Martin regorge d'informations intéressantes).

Il signifie:

1. **S**ingle Responsability principle
2. **O**pen-Close principle
3. **L**iskov substitution principle
4. **I**nterface segregation principle
5. **D**ependency inversion principle

Soit en Français:

1. **S**égrégation des interfaces
2. **O**uverture-Fermeture
3. **I**nversion de dépendances
4. **R**esponsabilité unique
5. **S**ubstitution de Liskov

. Ces cinq principes éprouvés sont considérés aujourd'hui comme fondamentaux pour une bonne conception objet.

Vous trouverez énormément de références sur internet qui les reprennent, mais parfois en passant à côté de certains détails ou subtilités qu'il convient de maîtriser. Je vous en propose donc une lecture commentée la plus exhaustive possible.

Le principe de responsabilité unique

Description

Une classe ne devrait avoir qu'une seule raison de changer

Autrement dit, si la modélisation d'une classe l'amène à couvrir une partie trop importante du sujet d'étude, elle risque de devenir difficile à maintenir car elle aura une responsabilité trop importante.

Pour rappel, la première responsabilité d'une classe est la gestion de ses attributs qu'elle est a priori censée encapsuler (pilier de la POO). Cela implique l'initialisation et/ou la construction des attributs, le fait de leur fournir une valeur par défaut, de gérer les valeurs non cohérentes, et éventuellement de les exposer (accesseurs et ou mutateurs).

Augmenter le nombre d'attributs d'une classe c'est donc augmenter ses responsabilités.

En second lieu, il est préférable de manipuler de petites classes bien faites, avec un bon compromis entre spécificité et généricité, afin d'en garantir la réutilisabilité dans d'autres contextes. Ainsi une classe trop spécifique risque de ne pas pouvoir être réutilisée, une classe trop générique de ne pas être utile.

Lorsqu'une classe ne respecte pas ce principe, on parle parfois de classe DIeu (elle connaît tout le monde, elle fait tout).

Comment respecter ce principe?

1. diviser pour mieux régner: il vaut mieux concevoir deux classes distinctes que tout faire dans une seule (voir exemple sur https://blog.codeinsider.fr/metal-gear-s-o-l-i-d/
2. éviter de gérer trop d'attributs: il est possible de les maintenir à l'aide de collections (un attribut collection au lieu de 20 attributs; pensez notamment aux MAP associatives).
3. déléguer: plutôt que de faire le travail dans la classe, externaliser tout ce qui peut l'être à l'aide de fabriques, wrappers, ou encore mieux des designs patterns
4. utiliser l'inversion de dépendance (voir plus bas) pour confier à ses héritiers l'implémentation d'une partie du code par exemple.

Le principe d'ouverture-fermeture

Description

Une classe doit être ouverte aux extensions mais fermées aux modifications

Principe tiré des travaux de Bertrand Meyer (Object-Oriented Software Construction, 1998).

Cela signifie qu'une bonne conception doit permettre à un ensemble de classes d'être réutilisées dans un logiciel qui sera en mesure de proposer des fonctionnalités nouvelles, basées sur ces classes, sans pour autant avoir à les modifier.

Il faut donc prévoir dans la conception que ces classes puissent s'adapter facilement et intrinsèquement à de nouveaux contextes.

Attention, *ce principe ne doit pas être confondu avec l'héritage*: l'héritage peut être une technique qui permet l'ouverture-fermeture, mais d'une part ce n'est pas toujours vrai (non respect du principe de Liskov par exemple) et d'autre part ce principe peut concerner le fonctionnement de plusieurs classes entre elles. L'héritage ne répondrait pas au problème dans ce cas.

Comment respecter ce principe?

Ce principe est très général, il convient donc de l'avoir en tête lorsque l'on conçoit une architecture.

Une première possibilité consiste à ***programmer des interfaces, pas des implémentations***. Autrement dit, s'assurer que la modélisation dispose d'un niveau d'abstraction suffisant (utilisation d'interfaces en respectant le principe de ségrégation, éviter les objets dieu difficilement extensibles, garantir la substitution de Liskov, etc.).

Ensuite, anticiper qu'une classe peut être dérivée: contraindre les héritiers à respecter le fonctionnement, exposer en public ou protected uniquement ce qui doit l'être, ne pas confier la gestion de ses attributs aux héritiers et plus généralement garantir l'encapsulation.

Enfin et plus généralement, garder ce principe en tête lorsqu'on ajoute une responsabilité à une classe, une nouvelle fonctionnalité ou un nouvel attribut. Ne pas oublier non plus que ce principe s'applique à l'architecture dans tout ou partie de son ensemble, et pas uniquement à chaque classe individuellement.

La substitution de Liskov

Description

Une abstraction A doit pouvoir être substituée par n'importe laquelle de ses sous-abstractions sans que cela n'affecte le programme

Autrement formulé, il s'agit lorsqu'on implémente des hiérarchies (extends) ou des comportements (implements en java) de s'assurer que la classe qui implémente réalise cette implémentation en

conservant la philosophie et les principes de fonctionnement de la classe de plus haut niveau d'abstraction.

Le fait d'exprimer un héritage ou une implémentation ne suffit donc pas à mettre en oeuvre le principe de substitution. Il s'agit bien de coder correctement la classe fille.

Prenons par exemple le code suivant:

```
public class LiskovError{

    private String message;

    public String toString(){
            String ret = "ok";

            setMessage("Affichage de l'objet dans la console");
            System.out.println(super.toString());
            return ret;
        }

    public void setMessage(String s){
            message = s;
        }
}
```

Quelles erreurs ont été commises?

1. la méthode toString() est censée renvoyer une chaîne de caractères décrivant l'instance. Ici, elle renvoie la chaîne “ok” uniquement. Si elle a des héritiers, ceux-ci ne pourront utiliser super.toString() sans enfreindre le comportement attentu de la méthode toString() tel que défini dans la classe Object
2. la méthode toString() n'est pas une méthode d'affichage: en faisant des System.out.println, en utilisant setMessage, elle prend donc de nouvelles responsabilités qui ne relèvent pas de son contrat de fonctionnement.

Une autre erreur fréquente vient d'une mauvaise modélisation d'une hiérarchie. L'exemple archi-classique est celui des formes géométriques (voir https://www.supinfo.com/articles/single/373-principe-substitution-liskov-lsp).

Comment mettre en œuvre ce principe

1. En respectant les contrats définis par les interface ou classe de haut niveau. **D'où l'importance de la documentation**! Si vous ne savez pas ce qu'est censé faire une méthode, vous pouvez la redéfinir et lui donner un comportement aberrant!
2. En utilisant des abstractions de haut niveau (interfaces en java, classes virtuelles pures, etc.) et en typant avec l'abstraction de plus haut niveau: vous pourrez ainsi substituer facilement une classe à une autre, pour peu que ces classes dérivent du type abstrait ET respectent le comportement défini dans l'abstraction.

La ségrégation des interfaces

Description

Voici un très bon article très pédagogique:
http://www.mechantblog.com/2014/02/solid-i-interface-segregation/.

Pensez à la réutilisabilité du code: trop spécifique amène à impossible à réutiliser.

Un bon concepteur se pose toujours la question de savoir s'il peut réutiliser un existant, et si ce n'est pas le cas comment rendre réutilisable ce qu'il doit développer.

Comment mettre en œuvre ce principe

- en évitant les abstractions dieu - en appliquant **aussi** le principe de responsabilité unique aux abstractions de haut niveau - en n'hésitant pas à recourir à la délégation et à la composition

L'inversion de dépendances

Description

Les classes (abstractions) de haut niveau ne doivent pas dépendre des classes de bas niveau (implémentations).

Ce principe illustre selon moi le mieux ce qu'est une conception objet.

Pour bien comprendre la notion de dépendance, vous pouvez lire cet article:
http://www.mechantblog.com/2014/05/solid-d-dependency-inversion/. **Attention**, il n'est pas exhaustif sur le principe d'inversion de dépendance mais donne un bon exemple pour se représenter les choses.

Comment mettre en œuvre ce principe

L'article précédent propose une manière simple d'obtenir une inversion de dépendance: passer par des abstractions de haut niveau.

On retrouve ce principe dans le pattern Observer/Observé par exemple.

Mais l'inversion de dépendance peut aller (beaucoup) plus loin. C'est vraiment l'idée de **ne pas se soucier des détails d'implémentation** qui prévaut. Autrement dit, si vous vous mettez en position de rendre vos classes de haut niveau fonctionnelles d'un point de vue logique, algorithmique et structurel, et que vous laissez bien le reste du travail à ceux qui feront vraiment le boulot (i.e. les classes concrètes), vous êtes dans de l'inversion de dépendance.

Un bel exemple est le DP template method: la classe abstraite code un algo non modifiable (méthode final), et quand elle n'est pas en mesure de fournir une partie du code, elle externalise cette partie

dans une méthode abstraite ou une interface que les héritiers auront en charge d'implémenter.

En conclusion, il ne faut pas hésiter en objet à dégrossir le travail et indiquer les grande lignes directrices dans les abstractions, faire en sorte que ces abstractions soient facilement interopérables à haut niveau, et déléguer aux classes concrètes (héritiers, implémenteurs) la responsabilité de fournir le code spécifique sans pour autant modifier le comportement établi dans les classes de haut niveau.

From:
http://bruno.mascret.fr/dokuwiki/ - **Bruno Mascret**

Permanent link:
http://bruno.mascret.fr/dokuwiki/doku.php/java:coo

Last update: **2020/05/18 11:16**