Table des matières

[Don't Repeat Yourself (DRY) 2](#__RefHeading___Toc3213_1093686814)

[You Aren't Gonna Need It (YAGNI) 4](#__RefHeading___Toc3215_1093686814)

[SOLID 5](#__RefHeading___Toc3217_1093686814)

[Single Responsibility Principle (SRP) 6](#__RefHeading___Toc3219_1093686814)

[Open Closed Principle (OCP) 7](#__RefHeading___Toc3243_1093686814)

[Liskov substitution principle (LSP) 10](#__RefHeading___Toc3245_1093686814)

[Interface Segregation Principle (ISP) 12](#__RefHeading___Toc3247_1093686814)

[Dependency Inversion Principle (DIP) 14](#__RefHeading___Toc3249_1093686814)

[Law of Demeter (LoD) (ou Principle of Least Knowledge) 16](#__RefHeading___Toc3251_1093686814)

[Soyez curieux 18](#__RefHeading___Toc1296_3794416847)

# Don't Repeat Yourself (DRY)

**Principe :**

Don't repeat yourself: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Ne vous répétez pas : Dans un système, toute connaissance doit avoir une représentation unique, non-ambiguë, faisant autorité.

**Historique :**

Andy Hunt et Dave Thomas dans leur livre « The Pragmatic Programmer » (1999).

**Comment faire :**

* code :
  + pas de copier/coller de code !
  + ne pas réimplémenter des logiques métier ou des utilitaires
  + refactoring pour factoriser les parties de code communes
* avoir un schéma de base de données normalisé
* générer du code (e.g. ce qui est fait par l’analyse)
* utiliser les annotations et/ou la réflexion (e.g. Lombok pour se débarrasser du code « boilerplate », par exemple des getters et setters qui dupliquent le fait qu’une classe a un attribut)

Andy Hunt et Dave Thomas parlent même de :

* plans de test
* système de build
* documentation

générés / maintenus automatiquement à partir d’une source de référence unique (e.g. JavaDoc généré à partir du code source).

**Rationnel :**

* Si la même logique ou données est présente à plusieurs endroits, il y a un coût à maintenir ces différentes implémentations. Souvent les implémentations finissent par diverger, ce qui signifie des bugs (effectifs ou potentiels).

**Cependant :**

Il existe des cas où on viole pertinemment DRY, par exemple en dénormalisant le schéma de la base de données pour des raisons de performance, mais il faut être conscient de cette duplication de logique ou de code et avoir clairement à l’esprit quelle est la représentation maître vs. la représentation esclave[[1]](#footnote-1).

**Références / exemples :**

* Quelques idées sur DRY : <http://wiki.c2.com/?DontRepeatYourself>

# You Aren't Gonna Need It (YAGNI)

**Principe :**

A programmer should not add functionality until deemed necessary.

Un programmeur ne doit pas ajouter une fonctionnalité avant qu'elle ne soit jugée nécessaire.

**Historique :**

Ce principe vient d'XP (Extreme Programming[[2]](#footnote-2)). Ron Jeffries a écrit : "Always implement things when you actually need them, never when you just foresee that you need them." (2001)

**Remarque :**

Tout ceci est applicable non seulement pour une fonctionnalité future, mais aussi pour une abstraction, une méthode, un paramètre de méthode... mis en place pour permettre une fonctionnalité future.

**Rationnel :**

* La fonctionnalité imaginée ne sera peut-être jamais réellement demandée.
  + On aura donc passé du temps à la concevoir, à l'implémenter et à la tester.
  + On aura complexifié le code (penser à KISS "Keep it simple, stupid").
  + On aura ajouté du code à maintenir.
  + On consommera plus de CPU, mémoire, espace disque, bande passante.. que nécessaire.

... pour rien.

* Même si la fonctionnalité imaginée est réellement demandée plus tard, cette demande sera vraisemblablement sensiblement différente de l'idée qu'on s'en fait aujourd'hui. Il faudra retravailler sur l'implémentation.
* Une autre fonctionnalité nécessaire maintenant sera livrée plus tard parce que, aujourd’hui, on a implémenté plus que nécessaire.

**Cependant :**

1) YAGNI only applies to capabilities built into the software to support a presumptive feature, it does not apply to effort to make the software easier to modify. (Martin Fowler)  
Ne pas utiliser YAGNI comme prétexte pour ne pas architecturer le code !

2) Dans certains cas, le coût futur de mettre en place la fonctionnalité plus tard est réellement beaucoup plus important que de le faire maintenant ; par exemple, le mécanisme de traduction, la gestion des droits...

**Références / exemples :**

* Une analyse rationnelle de l'intérêt du YAGNI : <https://martinfowler.com/bliki/Yagni.html>
* Quelques idées sur YAGNI : <http://wiki.c2.com/?YouArentGonnaNeedIt>

# SOLID

**5 principes :**

* Single responsibility - Responsabilité unique
* Open/closed - Ouvert/fermé
* Liskov substitution - Substitution de Liskov
* Interface segregation - Ségrégation des interfaces
* Dependency inversion - Inversion des dépendances

**Historique :**

Décrits avec d'autres principes dans des articles de « The C++ Report » (1996), puis dans un article "Design Principles and Design Patterns" de Robert C. Martin (2000). « Marketés » sous forme de l'acronyme SOLID par Michael Feathers.

**Références / exemples :**

* Les articles initiaux de Robert C. Martin (les exemples sont en C++, mais ils sont explicites même pour ceux qui ne connaissent pas le langage) :
  + <http://www.labri.fr/perso/clement/enseignements/ao/SRP.pdf>
  + <http://www.labri.fr/perso/clement/enseignements/ao/OCP.pdf>
  + <http://www.labri.fr/perso/clement/enseignements/ao/LSP.pdf>
  + <http://www.labri.fr/perso/clement/enseignements/ao/ISP.pdf>
  + <http://www.labri.fr/perso/clement/enseignements/ao/DIP.pdf>

## Single Responsibility Principle (SRP)

**Principe :**

A class should have only one reason to change.

(Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.)

Une classe ne doit changer que pour une seule raison.

(Chaque module ou classe devrait avoir une responsabilité sur une seule des fonctionnalités fournies par le logiciel et cette responsabilité devrait être entièrement encapsulée par la classe.)

**Historique :**

« Principle of cohesion » de Tom DeMarco dans son livre « Structured Analysis and System Specification » (1979).

**Rationnel :**

* Chaque responsabilité est une raison de changement. Une multitude de responsabilités dans une même classe aboutirait donc à une multitude de raisons de modifier cette classe et plus une classe a de raisons d’être modifiée moins elle est stable, la modification d'une responsabilité pouvant aboutir a un dysfonctionnement d'une autre responsabilité.  
  De plus, comme la classe a plusieurs responsabilités, il existe beaucoup de classes dépendantes de celles-ci. En cas de dysfonctionnement, l'impact peut donc être plus large.
* Si on utilise une classe que pour l'une de ses multiples responsabilités, on tire des dépendances qui ne devraient pas être présentes et plus il y a de dépendances, plus le code est difficile à maintenir.
* Quand on travaille sur la classe, on a besoin de garder plus de choses en tête (charge cognitive) que si la classe était réduite à une unique fonctionnalité.

**Un commentaire :**

* La notion d'« unicité » de raison de changement est relative : une classe Rectangle qui gère la géométrie et l'affichage peut être adéquate dans un logiciel qui ne fait que de l'affichage, mais elle aura trop de responsabilités pour un logiciel qui fait de l'affichage, de la manipulation de formes par l'utilisateur, des calculs de longueurs/surfaces …

**Références / exemples :**

## Open Closed Principle (OCP)

**Principe :**

We should write our modules/classes/... so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules/classes/... do, without changing the source code of the modules/classes/....

Nous devons écrire nos modules, classes,... de façon à pouvoir être étendus, sans avoir à les modifier. Autrement dit, nous voulons pouvoir changer ce que fait le module/classe/..., sans changer son code.

De façon plus pragmatique, nous voulons pouvoir ajouter des fonctionnalités en faisant un minimum de changements dans le code existant.

**Historique :**

Bertrand Meyer dans son livre « Object Oriented Software Construction » (1998) : « Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. »

**Un exemple** (extrait de <http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/>)**:**

public double Area(object[] shapes)

{

double area = 0;

foreach (var shape in shapes)

{

if (shape is Rectangle)

{

Rectangle rectangle = (Rectangle) shape;

area += rectangle.Width\*rectangle.Height;

}

else

{

Circle circle = (Circle)shape;

area += circle.Radius \* circle.Radius \* Math.PI;

}

}

return area;

}

n’est pas « closed » : si on ajoute une nouvelle forme géométrique, cette méthode devra être modifiée.

Un design correct serait :

public abstract class Shape

{

public abstract double Area();

}

public class Rectangle : Shape

{

public double Width { get; set; }

public double Height { get; set; }

public override double Area()

{

return Width\*Height;

}

}

public class Circle : Shape

{

public double Radius { get; set; }

public override double Area()

{

return Radius\*Radius\*Math.PI;

}

}

public double Area(Shape[] shapes)

{

double area = 0;

foreach (var shape in shapes)

{

area += shape.Area();

}

return area;

}

**Rationnel :**

* En limitant les changements dans le code existant, on limite le risque d'introduire une régression.

**Comment faire :**

* Au niveau d'un système complet : mécanisme de plugin (e.g. Eclipse)
* A niveau d'une classe :
  + Utiliser l'héritage pour étendre la classe de base dans la classe dérivée  
    Cependant attention à ne pas créer un couplage fort, ce qui arrive si la classe dérivée connaît les détails internes de la classe de base.
  + Utiliser le polymorphisme via des interfaces  
    cf les design patterns tels que [Decorator](http://kremer.cpsc.ucalgary.ca/patterns/decorator.html), [Strategy](http://kremer.cpsc.ucalgary.ca/patterns/strategy.html), [Visitor](http://kremer.cpsc.ucalgary.ca/patterns/visitor.html)...

**Un commentaire :**

* La notion de « fermeture » est relative : on ne peut par fermer une méthode, classe, module… contre tous les changements possibles. On ne prend en compte que les changements que l’on escompte (on retrouve ici YAGNI).

**Références / exemples :**

* <http://www.oodesign.com/open-close-principle.html>

## Liskov substitution principle (LSP)

**Principe :**

If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program (correctness, task performed...).

Si S est un sous-type (une classe dérivée) de T, alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme (justesse, tâche effectuée...).

**Historique :**

Barbara Liskov et Jeannette Wing dans leur article « Family Values: A Behavioral Notion of Subtyping » (1993) : « Let ϕ (x) be a property provable about objects x of type T. Then ϕ (y) should be true for objects y of type S where S is a subtype of T. »

**L’exemple toujours cité :**

Un carré n’est pas (au sens de l’héritage OO) un rectangle. Si setHeigth et setWidth sont « overridés » pour mettre tous deux height et width à la valeur de leur paramètre (pour assurer l’invariant height = width), alors on obtient un rectangle qui ne se comporte pas comme un rectangle, le test suivant n’est plus valide :

void testArea(Rectangle rect)

{

rect.setWidth(5);

rect.setHeight(4);

assert(rect.getArea() == 20);

}

**Rationnel :**

Si une classe dérivée S viole LSP, le code utilisant la classe T

* va fonctionner incorrectement lorsqu'il s'agit en fait d'une instance de S, ou
* devra être modifié pour connaître l’existence de la classe S et gérer celle-ci de façon particulière[[3]](#footnote-3).

**Techniquement cela signifie :**

* Pour chaque méthode « overridée » :
  + Contravariance (utiliser un type plus générique - moins dérivé - que celui spécifié dans T) des arguments dans S.
  + Covariance (utiliser un type plus spécifique - plus dérivé - que celui dans T) du type de retour dans S.
  + Aucune nouvelle exception ne doit être générée par la méthode de S, sauf si covariance (les exceptions générées sont des dérivées des exceptions générées par la méthode de T).
  + La visibilité doit être la même ou plus étendue.
* Pour la programmation par contrat[[4]](#footnote-4) :
  + Les préconditions (conditions vérifiées à l'entrée d'une méthode, notamment sur les paramètres) ne peuvent pas être renforcées dans une sous-classe, on ne peut pas avoir dans S des préconditions plus fortes que dans T.
  + Les postconditions (conditions vérifiées à la sortie d'une méthode, notamment sur la valeur de retour) ne peuvent pas être affaiblies dans une sous-classe, on ne peut pas avoir des postconditions plus faibles que dans T.
  + Les invariants (conditions vérifiées à l'entrée et à la sortie de toute méthode) de T sont respectés par les instances de S.
* Contrainte historique : Aucune méthode de S ne doit permettre des mutations interdites dans T.

**Références / exemples :**

## Interface Segregation Principle (ISP)

**Principe :**

No client should be forced to depend on methods it does not use.

Aucun client ne doit être forcé de dépendre de méthodes qu'il n'utilise pas.

En Java, cela signifie que lorsqu’on créé une interface, celle-ci ne doit pas contenir des méthodes qui correspondent à des responsabilités différentes. Il faut une interface par responsabilité.

**Historique :**

Robert C. Martin dans son article "The Interface Segregation Principle" (1996).

**Un exemple concret** (extrait de <https://academy.realm.io/posts/donn-felker-solid-part-4/>)**:**

public interface OnClickListener {

void onClick(View v);

void onLongClick(View v);

void onTouch(View v, MotionEvent event);

}

impose à un client qui n’est intéressé que par les clics d’écrire :

Button create = (Button)findViewById(R.id.create);

create.setOnClickListener(new View.OnClickListener {

public void onClick(View v) {

// assume this is a todo based app.

myDatabase.createTask(...);

}

public void onLongClick(View v) {

// do nothing, we're not worried about long clicking

}

public void onTouch(View v, MotionEvent event) {

// do nothing, we're not worried about touch

}

});

Il aurait fallu créer une interface par type d’événement.

**Rationnel :**

* C'est un cas spécifique du Single Responsibility Principe.
* La classe cliente qui implémente l'interface n’est pas obligée d'implémenter des méthodes non nécessaires.
* Un cas particulier du point précédent est quand on a à implémenter un mock.

**Références / exemples :**

## Dependency Inversion Principle (DIP)

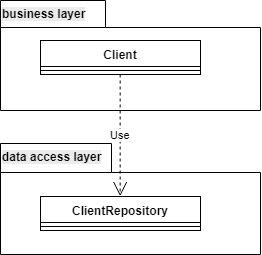
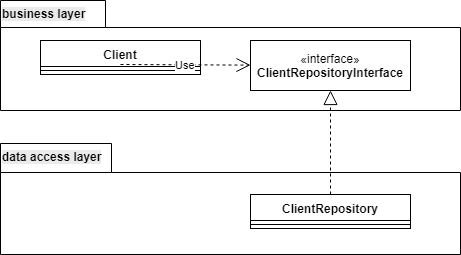
**Principe :**

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.
* Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
* Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

**Historique :**

Robert C. Martin dans son article "The Dependency Inversion Principle" (1996).

**Rationnel :**

* Classiquement, les couches de haut niveau dépendent des couches de bas niveau. Ceci rend les couches de haut niveau non réutilisables sans tirer les couches qui sont dessous.  
    
    
    
  L'inversion de dépendance consiste à rendre la couche de bas niveau dépendante d'une abstraction possédée par la couche de haut niveau.  
    
  
* L'interface définie par le niveau supérieur utilise la sémantique de celle-ci et ne définit que ce dont le niveau supérieur a besoin. On limite ainsi le couplage entre les deux niveaux et on évite ainsi que des détails techniques de la couche inférieure contamine le reste de l'application (par exemple que des types de données possédés par la BDD apparaissent dans la logique métier).
* Un cas important : Les tests unitaires. Ceci permet de mocker l'accès à la base de données en créant un mock qui implémente l'interface utilisée par la couche à tester.
* L'inversion de dépendance permet l'injection de dépendance (DI, Dependency Injection) : on définit par configuration quelles sont les implémentations de chaque interface (voir, par exemple, le Spring IoC container).
* L'inversion de dépendance permet à l'application de prendre le contrôle des choses qui lui sont importantes. Un exemple classique : pour une application dont la sémantique dépend beaucoup de la notion de date et heure courantes, on définit une interface pour y accéder, mais aussi pour pouvoir lui donner des valeurs et, ainsi, pourvoir faire des tests d'intégration plus simplement.

**Références / exemples :**

* Une bonne description de DI, IoC, DIP : <https://www.martinfowler.com/articles/dipInTheWild.html>

# Law of Demeter (LoD) (ou Principle of Least Knowledge)

*When you go to the store to purchase groceries, the clerk tells you the total cost. You then pull out your wallet, remove your credit card from it, and insert your credit card into the machine. Do you know what you don’t do? You don’t say to the clerk, “reach into my pants, pull out my wallet, get out the second credit card, and insert that into the machine.” Why not? Because it’s inappropriate coupling. It’s not the clerk’s job to know where you keep your* *wallet and where in your wallet you keep your credit card.*

**Principe :**

Only talk to your immediate friends.

Ne parlez qu'à vos amis immédiats.

En Java, un objet A peut appeler une méthode d'un objet B, mais A ne peut pas utiliser B pour accéder à un troisième objet et requérir ses services. Faire cela signifierait que A a une connaissance plus grande que nécessaire de la structure interne de B. Au lieu de cela, B peut être modifié si nécessaire pour que A puisse faire la requête directement à B, et B propagea la requête au composant ou sous-composant approprié. Si la loi est appliquée, seul B connaît sa propre structure interne.

Autrement dit :

Toute méthode M d'un objet O peut simplement invoquer les méthodes des types suivants d'objets :

* O lui-même
* les paramètres de M
* les objets que M crée/instancie
* les objets membres de O

En particulier, un objet doit éviter d'invoquer des méthodes d'un membre objet retourné par une autre méthode.

**Comment faire :**

La première solution qui vient à l’exprit est de remplacer

car.getOwner().getAddress().getStreet();

en implémentant :

public final class Car {

private final Owner owner;

public Street getOwnerAddressStreet() {

return owner.getAddressStreet();

}

...

}

Cela a le bénéfice de cacher (un peu) la structure interne de Car. Mais ce sera au prix d’une multiplication du nombre de méthodes.

Il faut surtout se demander pourquoi Car a besoin de la donnée Street. N’y a-t-il pas de la logique métier mise dans Car qui devrait plutôt être dans Owner ou Address ?

**Historique :**

Ian Holland (1987)

**Cependant :**

Ne pas tenir compte de cette règle lorsque les objets sont de pures structures de données (i.e. sans logique métier).

**Rationnel :**

* Les objets sont moins dépendants de la structure interne des autres objets, si celle-ci est changée, ils ne sont pas impactés.
* Ce principe pousse à mettre la logique métier dans les objets eux-mêmes au lieu d’avoir un objet orchestrateur qui connaît les détails internes de tous les objets qu’il manipule. (On retrouve ici une des bases de l’orientation objet…)

**Références / exemples :**

# Soyez curieux

Il existe d'autres principes :

* KISS (Keep it simple, stupid)
* Principle of least astonishment
* Express intent in your code
* Tell, Don't Ask
* Design Patterns
* Aggregation over inheritance
* Postel’s Law
* Beck's Design Rules[[5]](#footnote-5)
  + Passes the tests
  + Reveals intention
  + No duplication
  + Fewest elements
* ... etc

Il ne s'agit pas d'apprendre un maximum de règles et de les appliquer bêtement, mais de comprendre que les choix d'architecture (haut et bas niveau) sont des arbitrages selon certains critères, qui sont parfois contradictoires, et que la connaissance de ce que d'autres personnes conseillent est un moyen d'affûter son jugement.

1. Par exemple, dans le cas de l'application d'un PRA (plan de reprise d'activité) sur une base dénormalisée, quelle est la donnée maître qui va servir de référence pour initialiser ses copies esclave ? [↑](#footnote-ref-1)
2. <https://fr.wikipedia.org/wiki/Extreme_programming> [↑](#footnote-ref-2)
3. Ce qui est une violation du Open Closed Principe... [↑](#footnote-ref-3)
4. <https://fr.wikipedia.org/wiki/Programmation_par_contrat> [↑](#footnote-ref-4)
5. <https://martinfowler.com/bliki/BeckDesignRules.html> [↑](#footnote-ref-5)