**SOLID**

Par Michael Feathers en 2004 à partir des **principes** de programmation orientée objet identifiés par Robert Cecil Martin (Uncle Bob) en 2000 du livre [“Design Principles and Design Patterns.”](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf),

Ces **principes** visent à rendre le code plus lisible, facile à maintenir, extensible, réutilisable.

Il a été proposé comme un moyen de réfléchir spécifiquement à la qualité de la programmation orientée objet (OO). Dans l'ensemble, les principes SOLID présentent des arguments sur la façon dont le code doit être divisé, quelles parties doivent être internes ou exposées, et comment le code doit utiliser un autre code.

L'industrie est différente à bien des égards maintenant, mais certaines choses n'ont pas changé et ne changeront probablement pas. Ceux-ci inclus:

Le code est écrit et modifié par des personnes. Le code est écrit une fois et lu plusieurs fois. Il y aura toujours un besoin de code bien documenté, en particulier d'API bien documentées, qu'elles soient internes ou externes.

Le code est organisé en modules. Dans certaines langues, ce sont des classes. Dans d'autres, il peut s'agir de fichiers source individuels. Quoi qu'il en soit, il existe un moyen de séparer et d'organiser le code en unités distinctes et délimitées.

Le code peut être interne ou externe. Certains codes sont écrits pour être utilisés par vous-même ou votre équipe. D'autres codes sont écrits pour être utilisés par d'autres équipes ou même par des clients externes (via une API). Cela signifie qu'il doit y avoir un moyen de décider quel code est « visible » et ce qui est « caché ».

<https://stackoverflow.blog/2021/11/01/why-solid-principles-are-still-the-foundation-for-modern-software-architecture/>

**STUPID**

*Singleton : Couplage fort (sauf si utilisation de DI), non testable, global access!*

*Tight Coupling (couplage fort): créé à l’intérieur de la classe, intestable*

*Untestability: couplage, trop de dépendences, if imbriqués, bool dans les paramètres…*

*Premature Optimizations: ou codegolf*

*Indescriptive Naming: for(i), if $n… -> no comments, use function name as comments*

*Duplication: copy/paste*

- - -

**Single-responsibility Principle (SRP)** : Cohesion and coupling

A class should have one and only one reason to change, meaning that a class should have only one job

*A class should have one, and only one, reason to change.* Robert C. Martin.

The SRP is one of the simplest of the principles, and one of the hardest to get right.

*RCM: What defines a reason to change? Si tu amène ta voiture au garage pour changer le moteur, et tu te retrouves avec les essuis-glaces qui ne fonctionnent plus.*

*This principle is about people. Qui maintient la charge du code de la classe? Quelle équipe?*

What is Responsability? If you can think of more than one motive for changing a class, then that class has more than one responsibility.

Decouple depending on the context, or you might add unnecessary complexity

**cohesion** refers to the *degree to which the elements inside a* [*module*](https://en.wikipedia.org/wiki/Module_(programming)) *belong together*

* *The functionalities embedded in a class, accessed through its methods, have much in common.*
* *Methods carry out a small number of related activities, by avoiding* [*coarsely grained*](https://en.wikipedia.org/wiki/Granularity#Data_granularity) *or unrelated sets of data.*
* *Related methods are in the same source file or otherwise grouped together; for example, in separate files but in the same sub-directory/folder.*

- - -

**Open/Closed Principle (OCP) : encapsulation**

Objects or entities should be open for extension but closed for modification.

[Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer) (Eiffel)

*Robert C. Martin considered this principle as the “the most important principle of object-oriented design”.* Bertrand Meyer wrote about it in 1988, with inheritance.

“*A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.*”

That’s why Robert C. Martin and others redefined the Open/Closed Principle to the Polymorphic Open/Closed Principle. Interfaces introduce additional level of abstraction which enables loose coupling.

*A module will be said to be open if it is available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.*

*You should be able to extend a class behavior, without modifying it. Robert C Martin*

1 . inheritance

2. Interface

3. Plugins

*“For Martin, as we saw above, the “plugin architecture” is the apotheosis of the OCP: What if the design of your systems was based around plugins, like Vim, Emacs, Minecraft, or Eclipse?”*

-> “Strategy” design pattern

-> like “Closure” functions

-> Data driven creation

-> members variables must be “private” : encapsulation

-> ex: array\_merge(), will works with every type of data like an array, without changing its code

*Plugin systems are the ultimate consummation, the apotheosis, of the Open-Closed Principle. R.C M*

A module will be said to be closed if is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

A module will be said to be open if it is available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

- - -

**Liskov Substitution Principle (LSP) :** Barbara Liskov

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

*Derived classes must be substitutable for their base classes.* Robert C. Martin.

“Principe de moindre surprise” - “Code correctness”

One is able to ensure that any potential call to a base class that ends up being a derived class is not broken, due to providing a more generic input than the method was expecting and that it is not able to deduce its type. On the other way around, if return outputs are more generic, whatever use of the output of a base class your component was using, it still will be able to use it

*Le principe de substitution de Liskov indique qu'il doit être possible pour un objet de type T acceptant une dépendance de type S, de pouvoir remplacer cette dernière par une dépendance d'un type dérivé de S sans que cela ait le moindre impact sur le fonctionnement du code.*

*-> Each interface an intention-revealing name made the code easier*

*-> Don’t add Exceptions*

**Voici les 3 conditions a remplir pour être conforme au Liskov’s Substitution Principle.**

* [Contravariance](https://fr.wikipedia.org/wiki/Covariant_et_contravariant_(alg%C3%A8bre_lin%C3%A9aire)#Autres_usages_du_vocable) des arguments de méthode dans le sous-type.
* [Covariance](https://fr.wikipedia.org/wiki/Covariant_et_contravariant_(alg%C3%A8bre_lin%C3%A9aire)#Autres_usages_du_vocable) du type de retour dans le sous-type.
* Aucune nouvelle [exception](https://fr.wikipedia.org/wiki/Exception_(informatique)) ne doit être générée par la méthode du sous-type, sauf si celles-ci sont elles-mêmes des sous-types des exceptions levées par la méthode du supertype.

<https://blog.codavel.com/2018/09/25/liskov-substitution-principle>

First and foremost: It ensures that you are getting your abstractions right. The whole point of being able to create a class based on an abstract class, is that you can use the derived class seamlessly, without having to touch any of the other derived classes and without breaking any builds.

Second reason: It’s an excepcional tool for detecting poor designs. For instance, if you have an abstraction and one of your derived classes introduces a field or method that requires special treatment (“an if clause” for early return), you’ll probably be in violation of the LSP (you are breaking the contract with the expected behaviour of the abstract class). If this is the case, you should definitely revisit your abstraction!

Third reason: Your unit test code and integration code for the base class is fully reusable for your derived classes. This is highly relevant, since in any cases, unit tests are something that is very time-consuming, so if you can reuse that code, you may save a lot of time!

Fourth reason (this one is particularly nice for me!): If you’re doing some sort of comparison or A/B testing of software components that must have the same behaviour, it allows you to easily replace these components, even on the fly!

- - -

**Interface Segregation Principle (ISP)** : (like SRP for interfaces)

A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.

*Make fine grained interfaces that are client specific.* Robert C. Martin.

*Don’t show your clients more than they need to see”.*

- - -

**Dependency Inversion Principle (DIP) :** <> dependency injection

Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

High level is using Low-level

*Depend on abstractions, not on implementations.* Robert C. Martin.

Inversion of Control: Hollywood law: Don’t call me, I’ll call you

<https://martinfowler.com/articles/dipInTheWild.html>

**REF**

* <https://afsy.fr/avent/2013/02-principes-stupid-solid-poo>
* <https://stackify.com/solid-design-principles/>

OCP

* <https://thevaluable.dev/open-closed-principle-revisited/>
* <https://dev.to/tamerlang/understanding-solid-principles-open-closed-principle-5e25>

[https://reflectoring.io/open-closed-principle-explained](https://reflectoring.io/open-closed-principle-explained/)

Robert C Martin

* <https://blog.cleancoder.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html>
* <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>

- - -

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

—[*Refactoring*](https://martinfowler.com/books/refactoring.html)*, Martin Fowler with Kent Beck, 1996*

- - -

**Against:**

* Trop vague, pas adapté
* SRP: Pointlessly Vague Principle
* OCP: *Change the code to make it do something else!*
* LSP: Trop complexe: juste “Principle of Least Surprise”
* ISP: it’s just that it isn’t a principle, it is a pattern. A principle is something that is generally good advice in any context: A pattern is a strategy that works in a given context
* DIP: A dependency is only interesting when there might be multiple ways of providing it,, Most dependencies don’t need inverting
* <https://dannorth.net/2022/02/10/cupid-for-joyful-coding/>
* [**Composable**](https://dannorth.net/2022/02/10/cupid-for-joyful-coding/#composable)**:** plays well with others
* [**Unix philosophy**](https://dannorth.net/2022/02/10/cupid-for-joyful-coding/#unix-philosophy)**:** does one thing well
* [**Predictable**](https://dannorth.net/2022/02/10/cupid-for-joyful-coding/#predictable)**:** does what you expect
* [**Idiomatic**](https://dannorth.net/2022/02/10/cupid-for-joyful-coding/#idiomatic)**:** feels natural
* [**Domain-based**](https://dannorth.net/2022/02/10/cupid-for-joyful-coding/#domain-based)**:** the solution domain models the problem domain in language and structure

Ex: https://alexsoyes.com/solid/