

A close-up photograph of a brown goat with dark brown patches on its back and legs. The goat is tied to a vertical wooden post with a thick, light-colored rope. It is standing on a ground covered with dry, brown soil and some scattered debris. In the background, there are more dry plants and a faint view of a green hillside.

# *Principes – SOLID*

## *De STUPID à SOLID*

# *Objectifs*

- Pratiques « STUPID » à éviter
- Principes SOLID



# *STUPID ?*

- S : Singleton
- T : Tight coupling
- U : Untestability
- P : Premature optimisation
- I : Indescriptive naming
- D : Duplication

# *STUPID - Singleton*

- Pensez à un objet partagé par tout un programme (static)
  - Effet de bord
  - Les programmes qui utilisent un état global cachent leurs dépendances
  - Tests unitaires très difficiles à effectuer (car liés à un état global)



# *STUPID - Tight coupling*

- Le degré de couplage est dépendant du nombre de classes dont une classe dépend :
  - Héritage
  - Types de données membres
  - Types utilisés en paramètres
  - Types utilisés localement dans les méthodes
- On cherche à réduire le couplage entre les méthodes, les classes et les modules de manière générale (package, namespace, librairies, etc.)



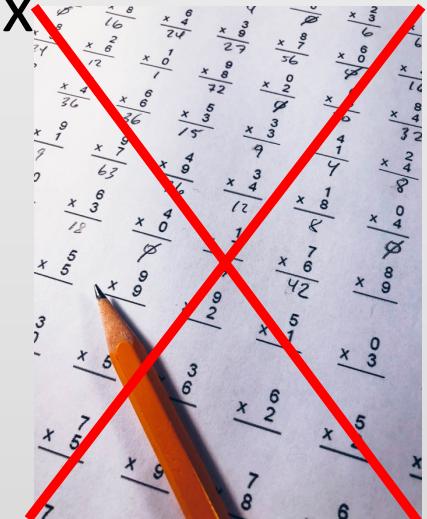
# *STUPID - Untestability*

- Le code est non testable ou difficilement testable quand le couplage est trop fort : il faut alors travailler beaucoup pour monter un test
- Signe d'un mauvais code et d'une mauvaise conception



# *STUPID - Premature optimisation*

- Ne faites pas d'optimisation prématuée de votre code, vous ne savez pas comment il va être utilisé, comment il va évoluer
- Une optimisation coûte cher et n'a pas forcément les bénéfices attendues et n'équilibre souvent pas le cout
- L'optimisation est souvent plus complexe que le choix d'un type de donnée ou un type de boucle. Elles sont souvent plus liées aux interactions avec les autres systèmes.



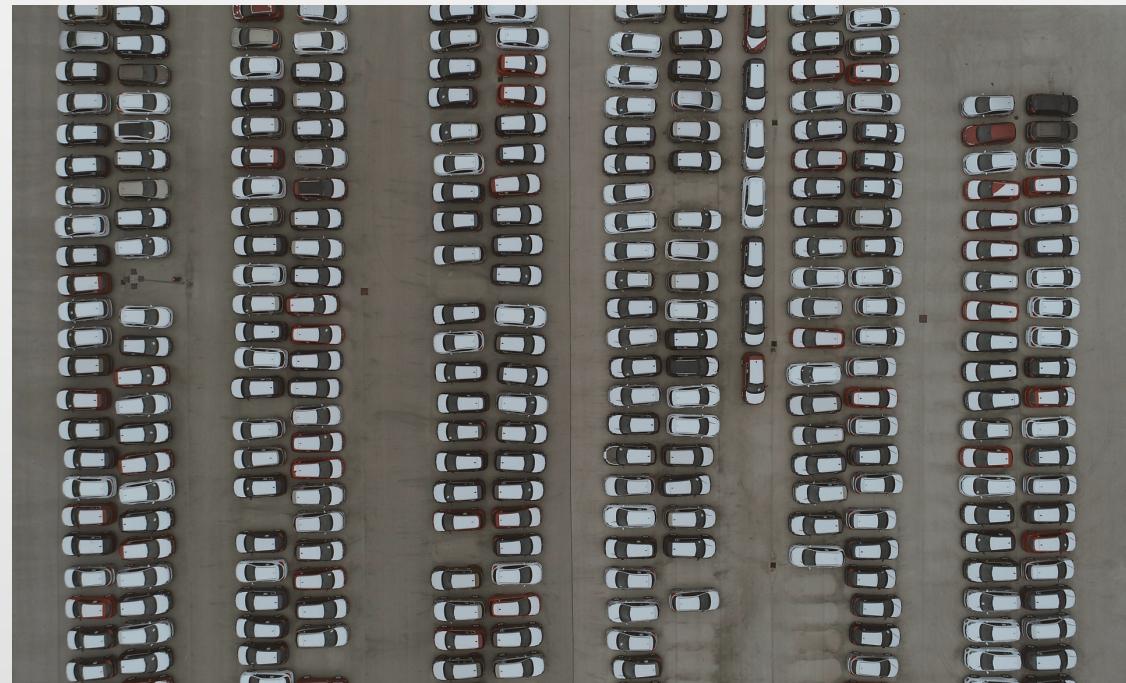
# *STUPID - Indescriptive naming*

- Nommez vos classes, méthodes, etc. avec des noms évocateurs, qui ont du sens
- Évitez les abréviations, vous vous adressez à des humains pas à une machine
- Évitez les valeurs magiques, donnez du sens à ces valeurs en créant une constante qui a un nom significatif



# *STUPID - Duplication*

- La duplication de code est dangereuse
- La duplication de connaissances l'est aussi (voir cours de bases de données)
- Soyez un bon développeur en appliquant :
  - DRY : Don't Repeat Yourself
  - KISS : Keep It Sweet and Simple
- Appliquez la règle du moindre effort mais de la bonne façon

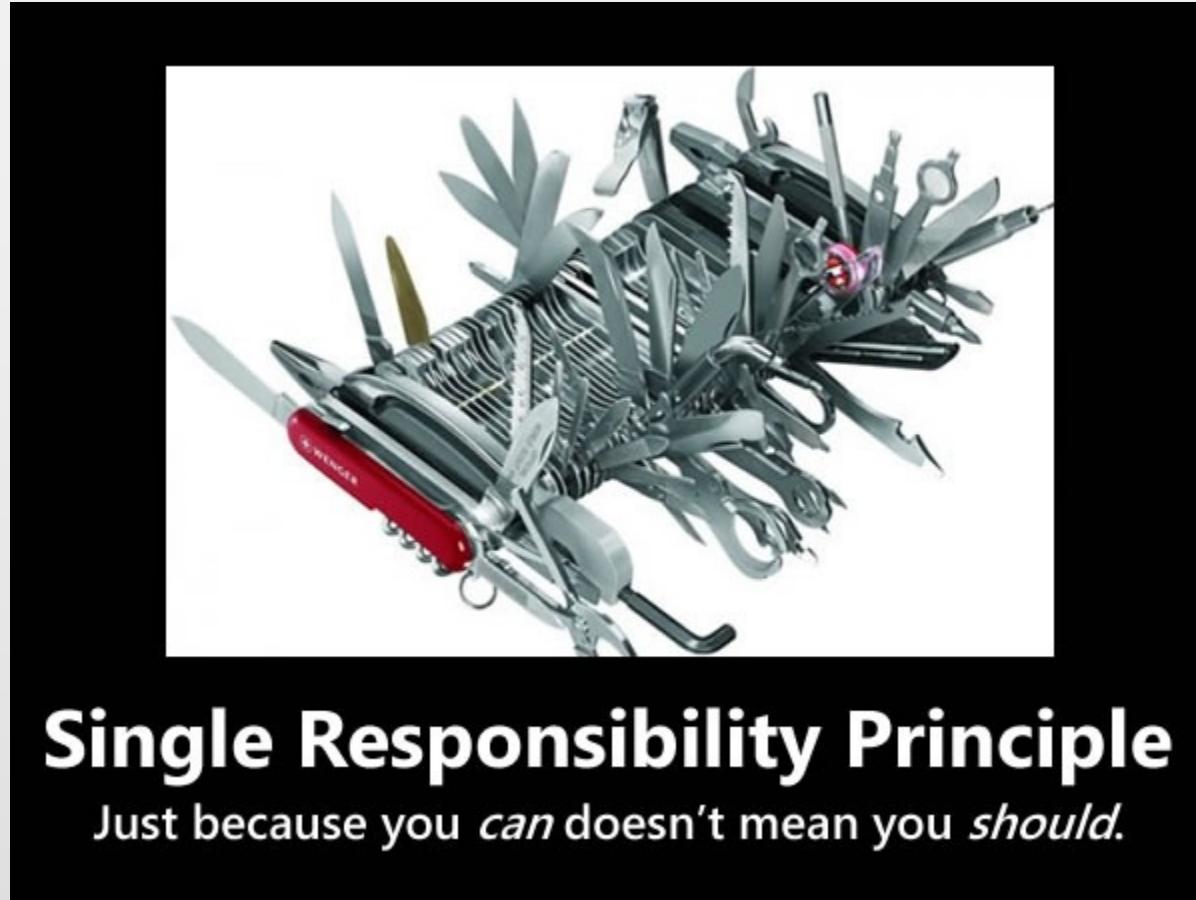


# SOLID ?

- Principes introduits par Michael Feathers et Robert C. Martin
  - S : Single Responsibility Principle (SRP)
  - O : Open/close principle (OCP)
  - L : Liskov substitution principle (LSP)
  - I : Interface segregation principle (ISP)
  - D : Dependency inversion principle (DIP)



# *Single Responsibility Principle - SRP*



# *Single Responsability Principle - SRP*



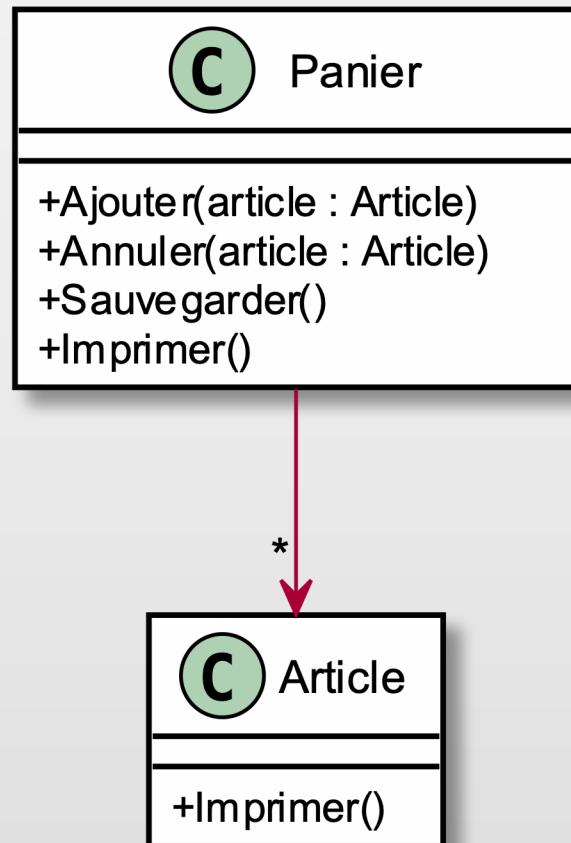
**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.

- = Responsabilité unique
- But : rendre la classe plus robuste
- Une classe, une fonction ou une méthode doit avoir seulement une responsabilité
- La classe doit être cohérente : si son nom est cohérent avec sa fonctionnalité, le principe devrait être respecté
- Robert C. Martin dit « une classe ne doit changer que pour une seule raison »
- Contre exemple : une classe à les responsabilités de compiler et d'imprimer un rapport. Ici, la classe peut changer pour deux raisons :
  - Le contenu peut changer (calcul des taxes, promotions, etc.)
  - Le format du rapport peut aussi changer (graphismes, supports : papier/web/courriel, etc.)

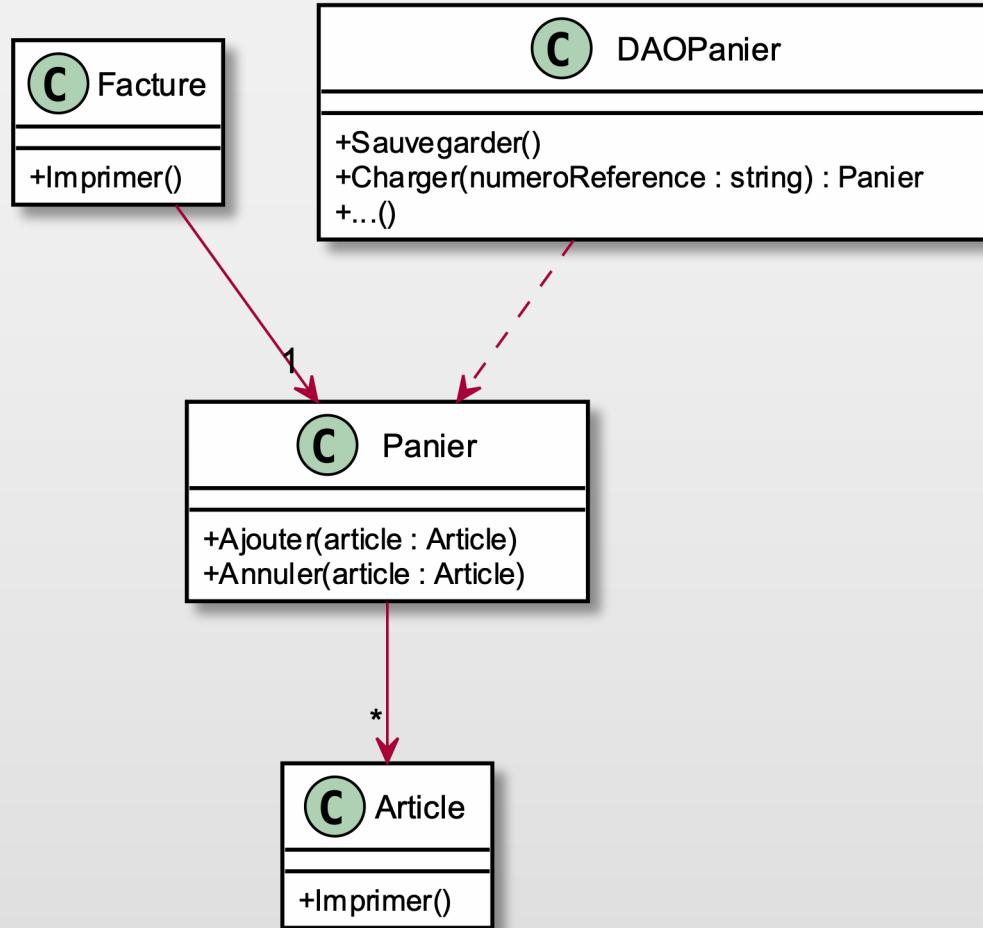
# SRP – exemple 1 de violation



**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.



# SRP – exemple 1 solution possible

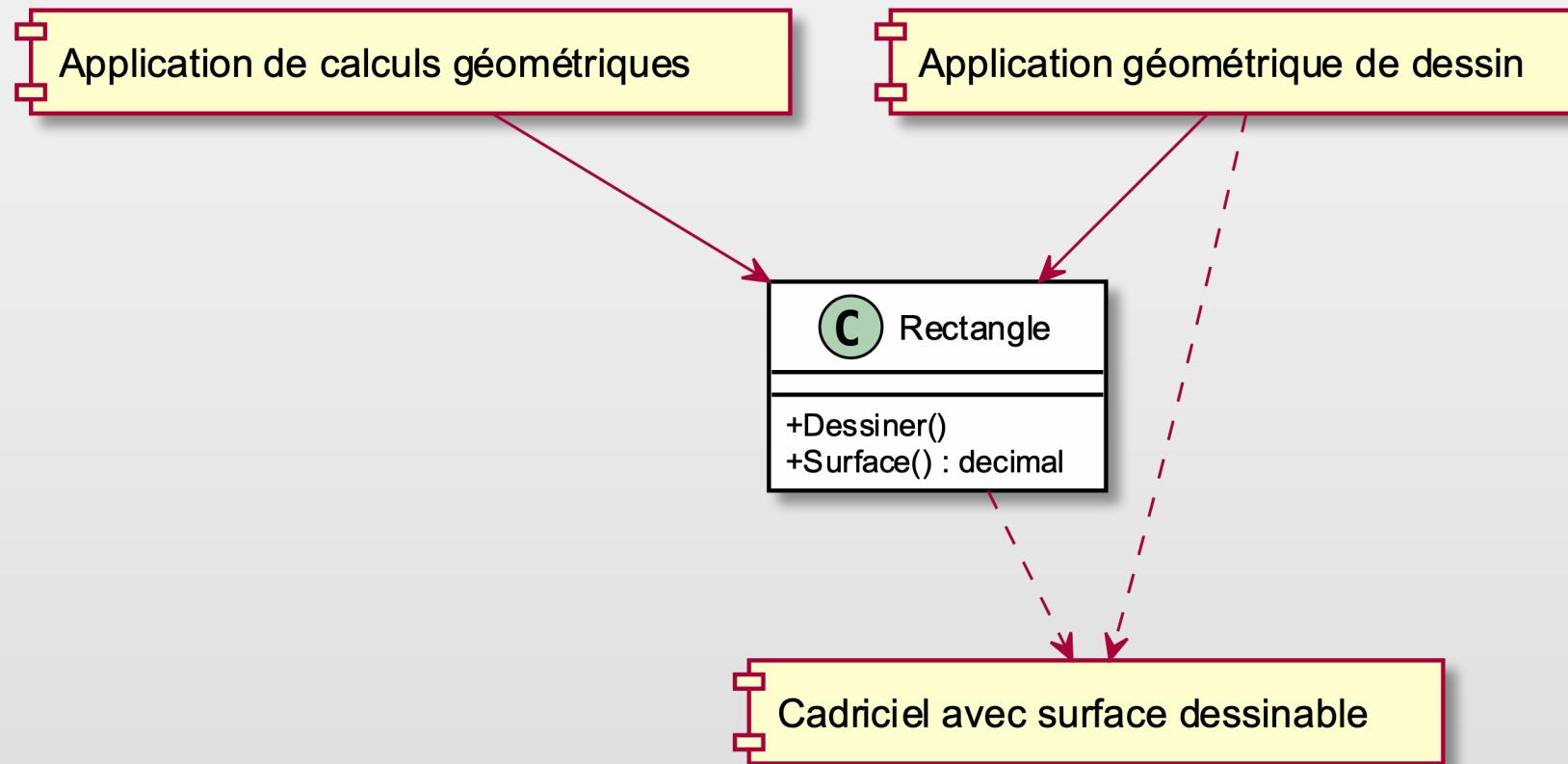


**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.

# SRP – exemple 2 de violation



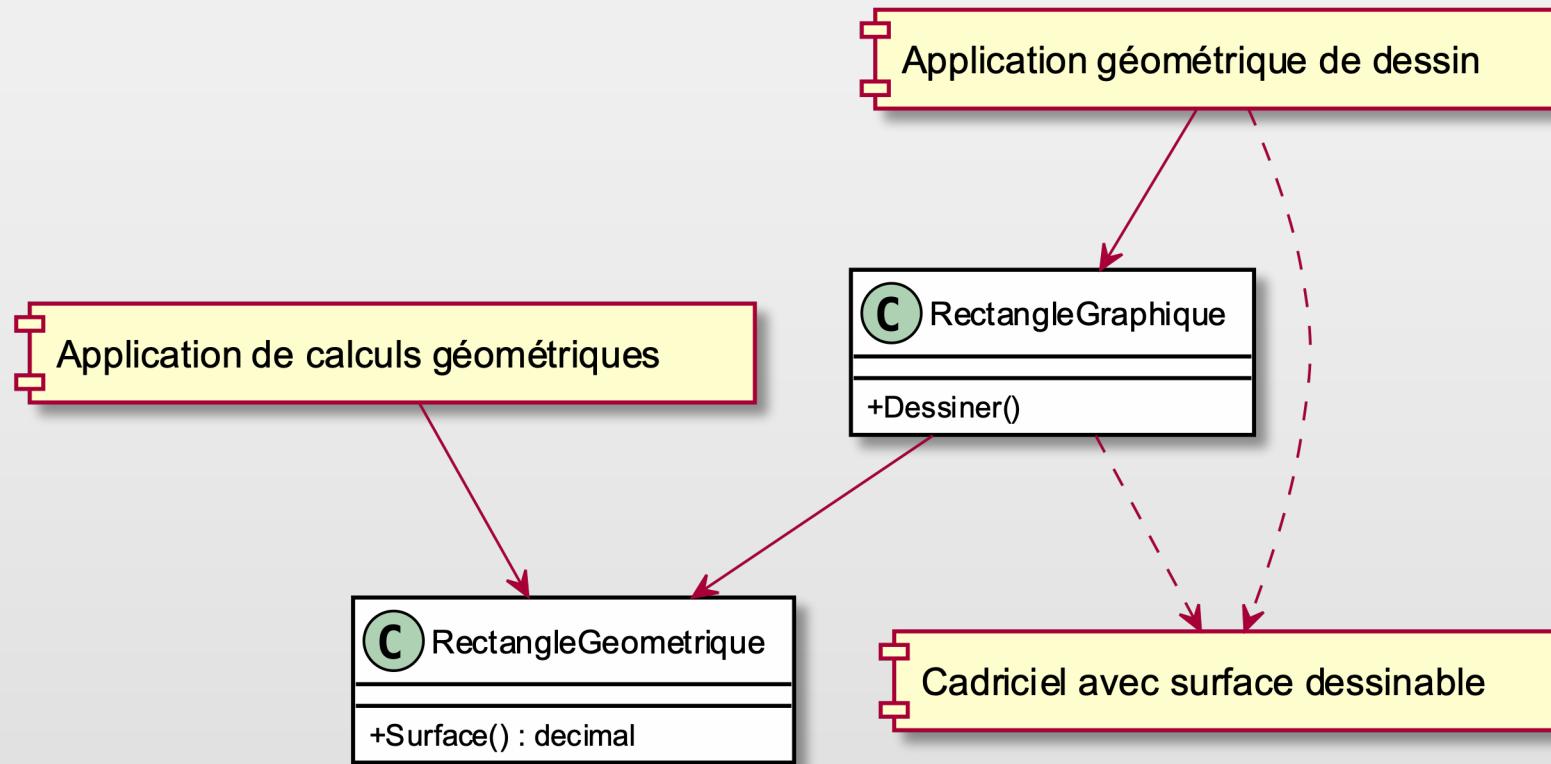
**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.



# SRP – exemple 2 solution possible

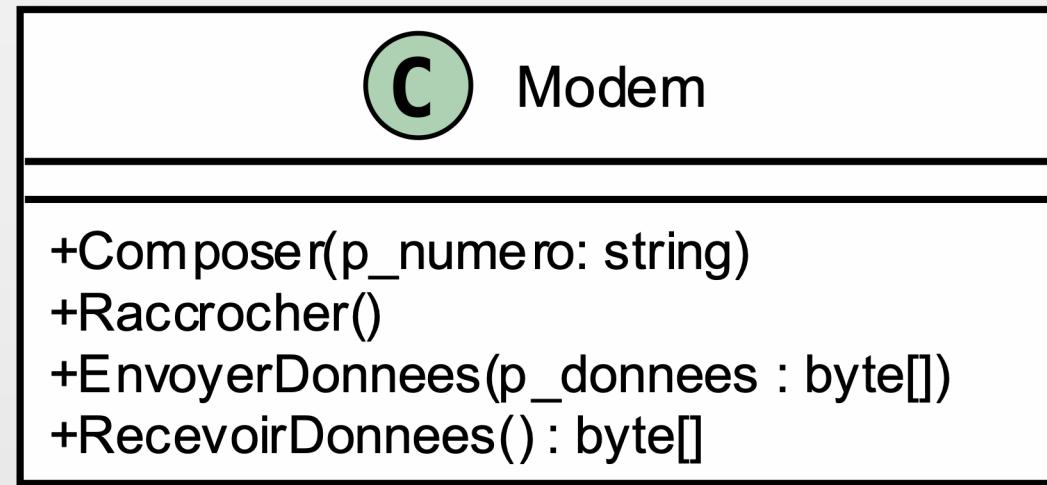


**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.



# SRP – exemple 3 de violation

Exercice : Comment améliorer le diagramme ?



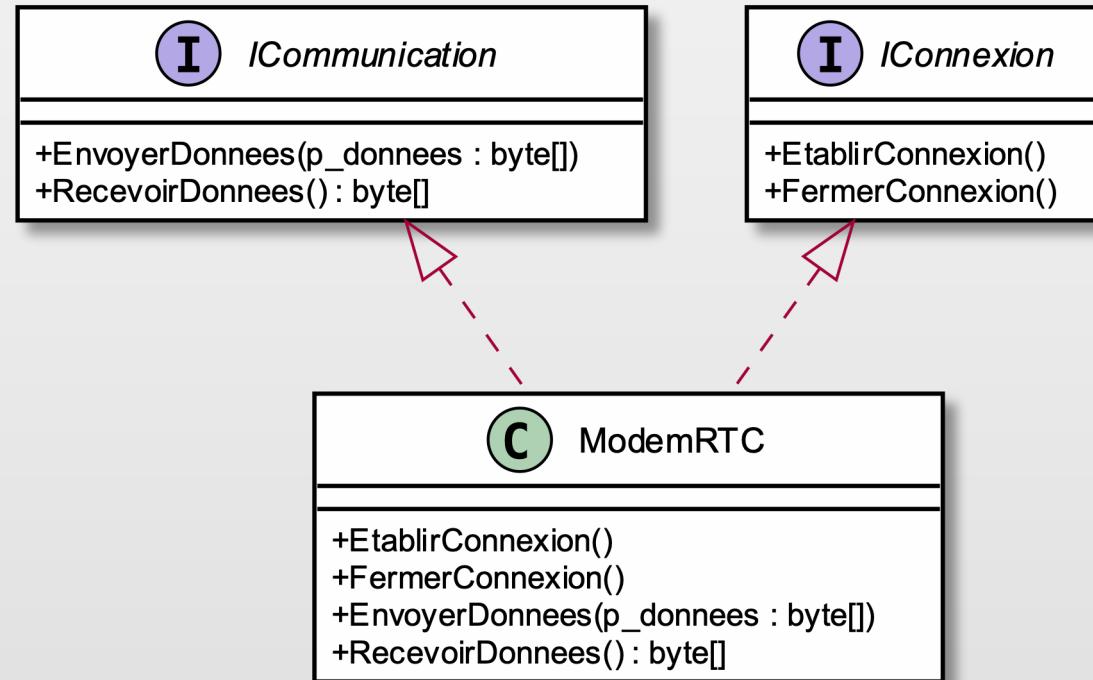
**Single Responsibility Principle**  
Just because you *can* doesn't mean you *should*.

# SRP – exemple 3 solution possible



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.



# *Open/close principle – OCP*



# *Open/close principle – OCP*



**Open-Closed Principle**

Open-chest surgery isn't needed when putting on a coat.

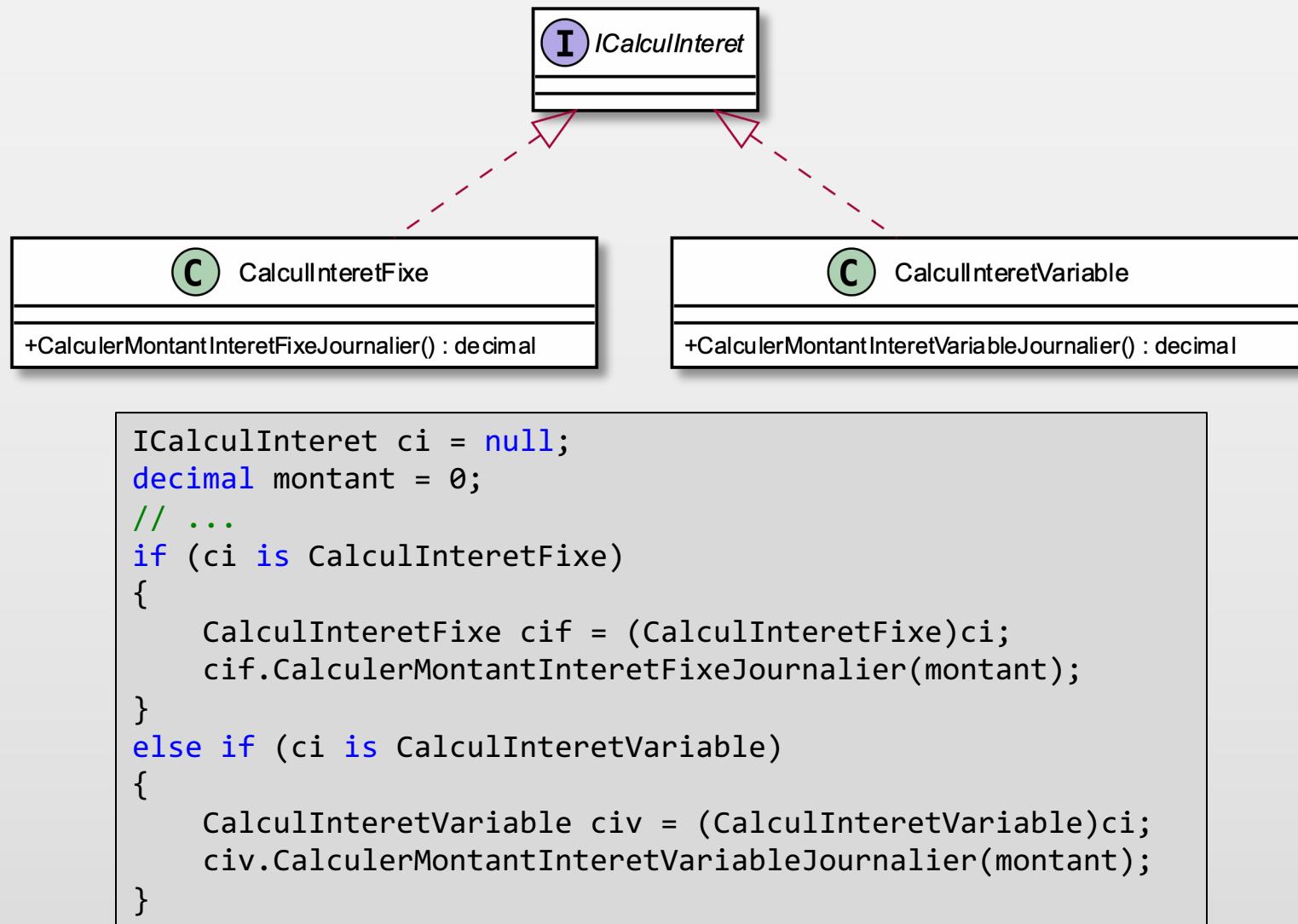
- = Ouvert / fermé
- Une classe doit être ouverte à l'extension mais fermée aux modifications
- L'idée est ici d'interdire la modification d'une classe une fois qu'elle est livrée/acceptée (sauf en cas de bug)
- L'ajout de fonctionnalités se passe par extension  
⇒utilisation des abstractions et du polymorphisme
- Principe très difficile à respecter dans la réalité : il faut prévoir les changements (?)
- Peut entraîner une complexité inutile

# OCP – Violation #1



**Open-Closed Principle**

Open-chest surgery isn't needed when putting on a coat.

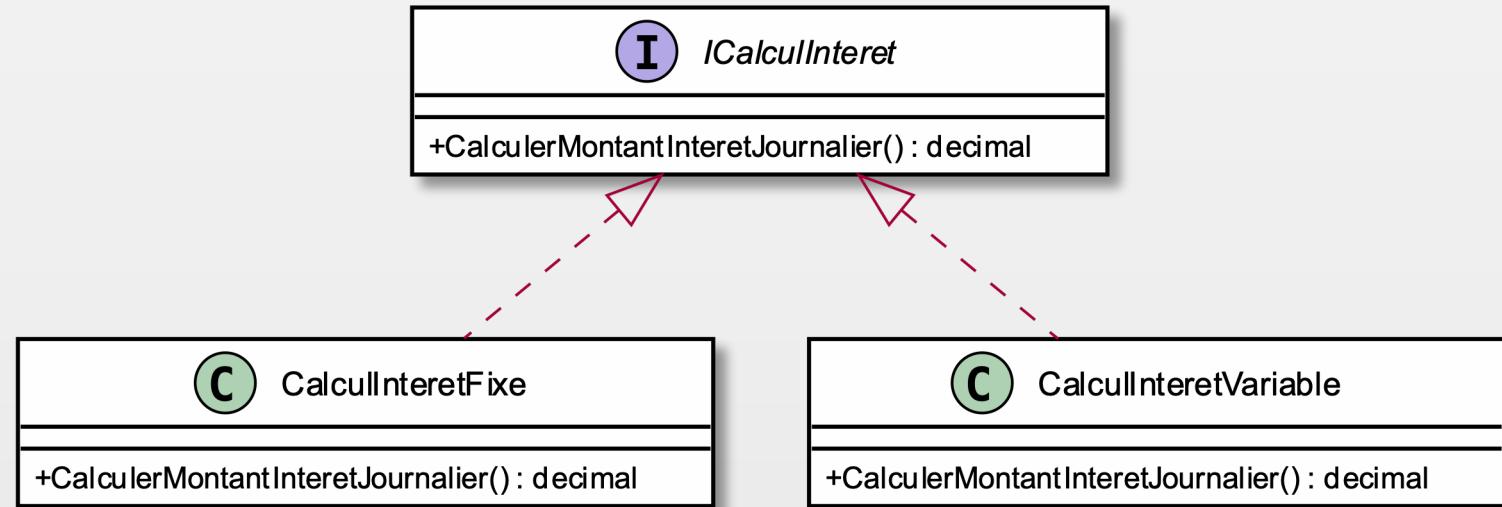


# OCP – Résolution #1



**Open-Closed Principle**

Open-chest surgery isn't needed when putting on a coat.



```
// ...
ICalculInteretV2 ci = null;
decimal montant = 0;
// ...
ci.CalculerMontantInteretJournalier(montant);
```

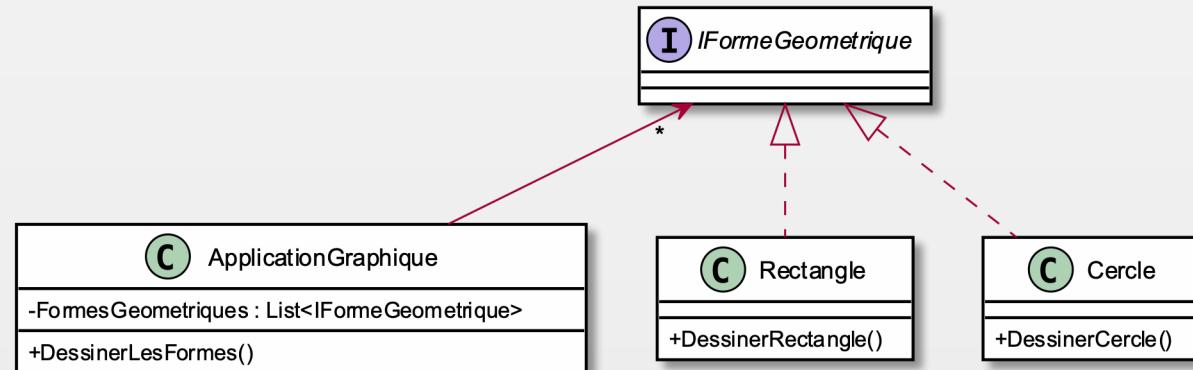
# OCP – Violation #2



**Open-Closed Principle**

Open-chest surgery isn't needed when putting on a coat.

Exercice : Comment améliorer le diagramme et le code ?



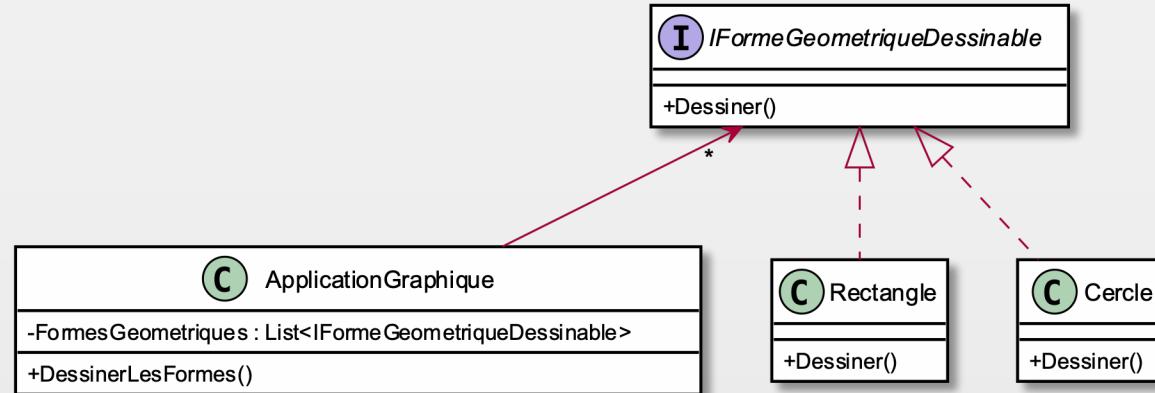
```
// ...
foreach (IFormeGeometrique formeGeometrique in this.FormesGeometriques)
{
    switch (formeGeometrique)
    {
        case Rectangle r:
            r.DessinerRectangle();
            break;
        case Cercle c:
            c.DessinerCercle();
            break;
        default:
            break;
    }
}
// ...
```

# OCP – Résolution #2



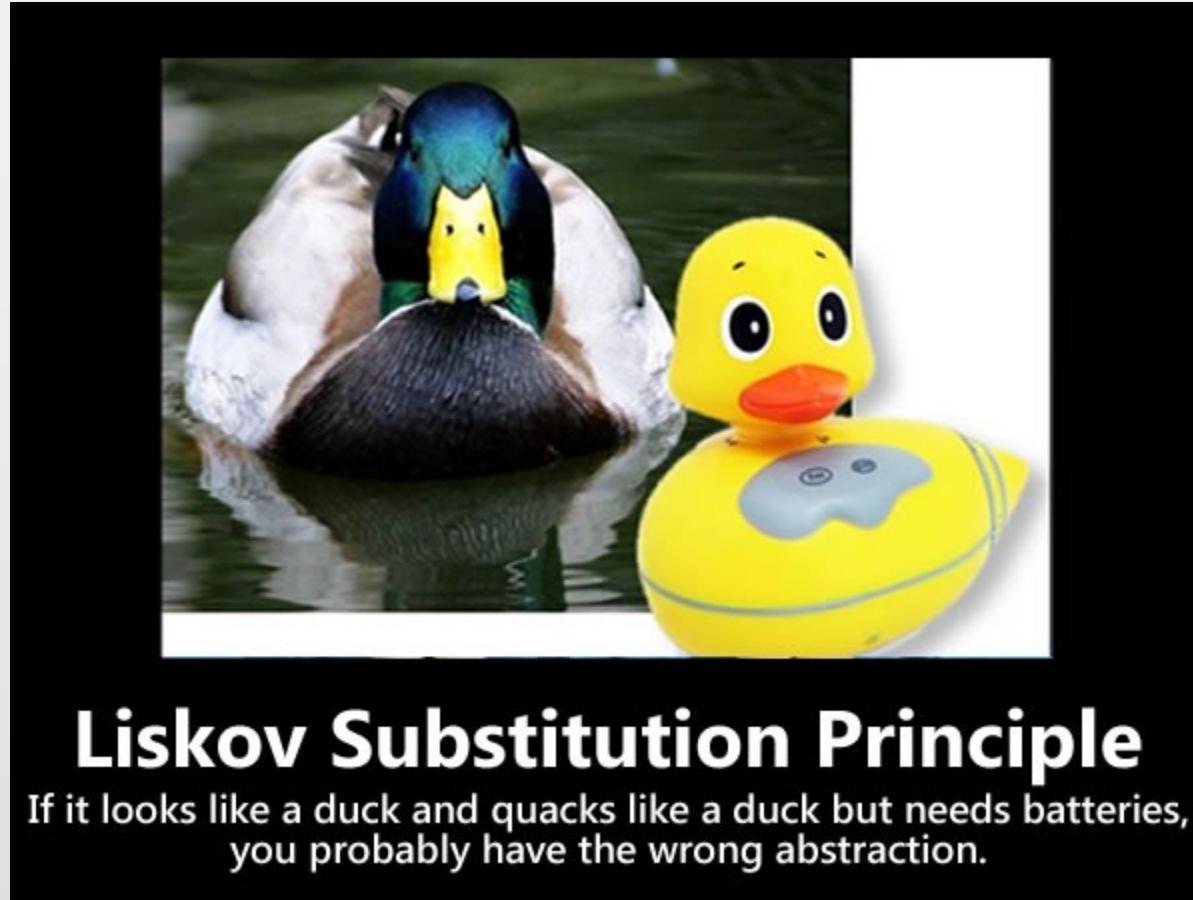
**Open-Closed Principle**

Open-chest surgery isn't needed when putting on a coat.

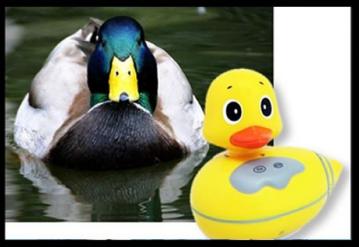


```
// ...
foreach (IFormeGeometrique formeGeometrique in this.FormesGeometriques)
{
    formeGeometrique.Dessiner();
}
// ...
```

# *Liskov substitution principle – LSP*



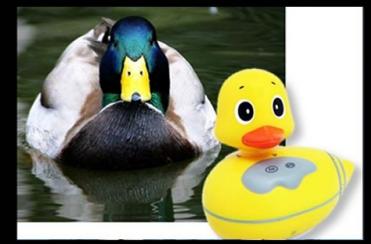
# *Liskov substitution principle – LSP*



**Liskov Substitution Principle**  
If it looks like a duck and quacks like a duck but needs batteries,  
you probably have the wrong abstraction.

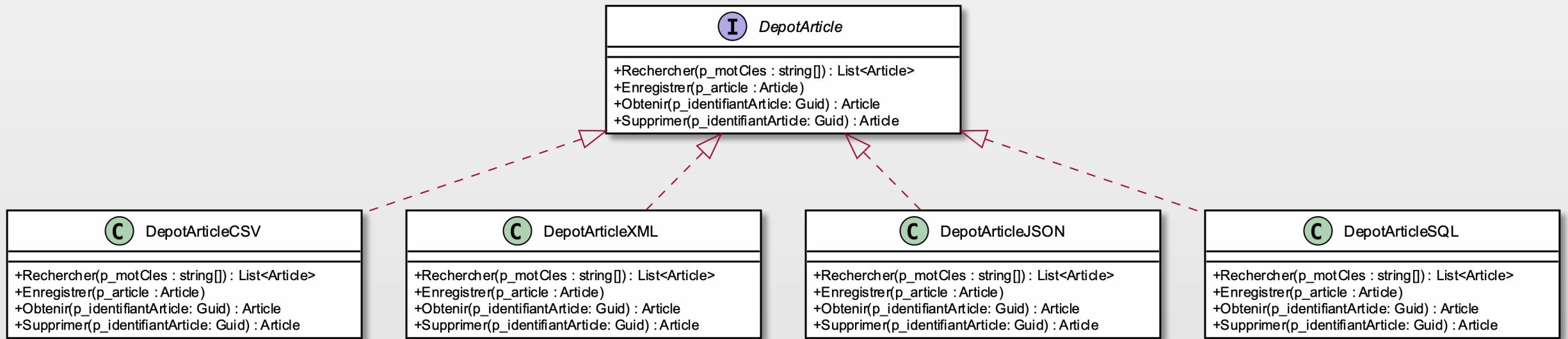
- = Substitution de Liskov
- Une instance du type M doit pouvoir être substituée par une instance du type F tel que F est dans la hiérarchie d'héritage de M (Sous-type direct ou non)
- La classe F doit avoir un **comportement similaire** à ce qui est attendu par les utilisateurs de la classe M
  - Des classes qui utilisent M ne doivent pas avoir conscience d'utiliser F
- La classe F ne doit pas en faire moins que la classe M
- L'utilisation d' » instanceof » est suspecte

# LSP – Exemple

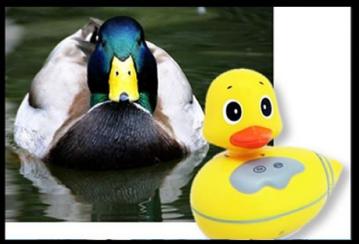


**Liskov Substitution Principle**

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.



# LSP – Violation le grand classique

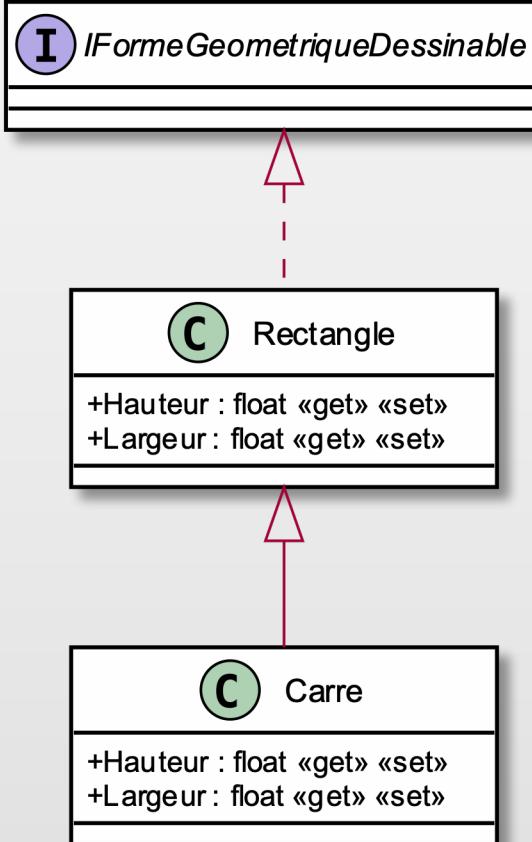


**Liskov Substitution Principle**

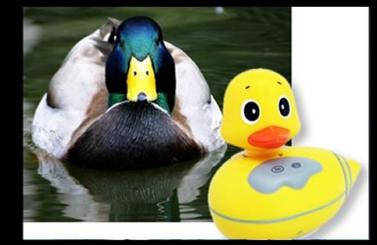
If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

Exercice :

- Est-ce que le Carré se comporte comme le Rectangle ici ?
- Écrire le code des propriétés Hauteur et Largeur de Rectangle et Carré
- Écrire un bout de code qui montre que le LSP n'est pas respecté
- Que dire du calcul de la surface ?

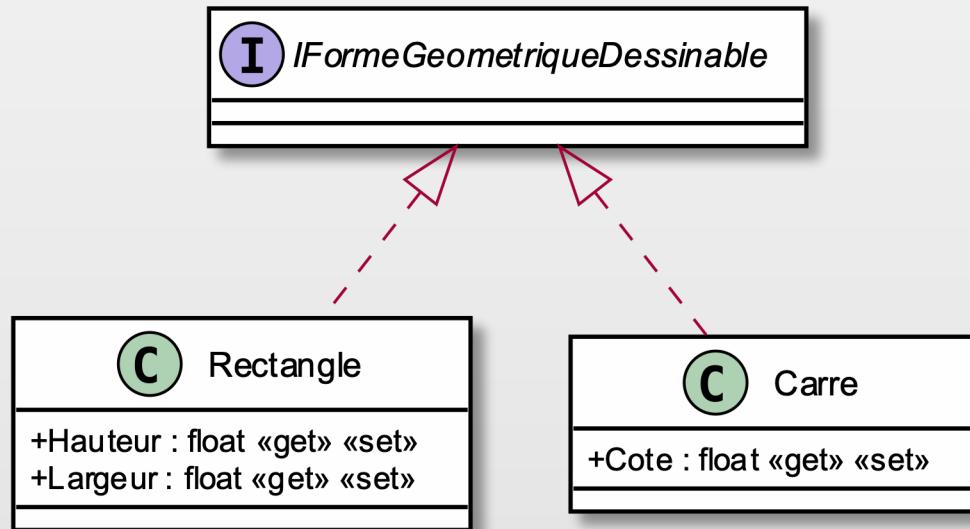


# LSP – Violation le grand classique - Résolution



**Liskov Substitution Principle**

If it looks like a duck and quacks like a duck but needs batteries,  
you probably have the wrong abstraction.



# *Interface segregation principle ISP*



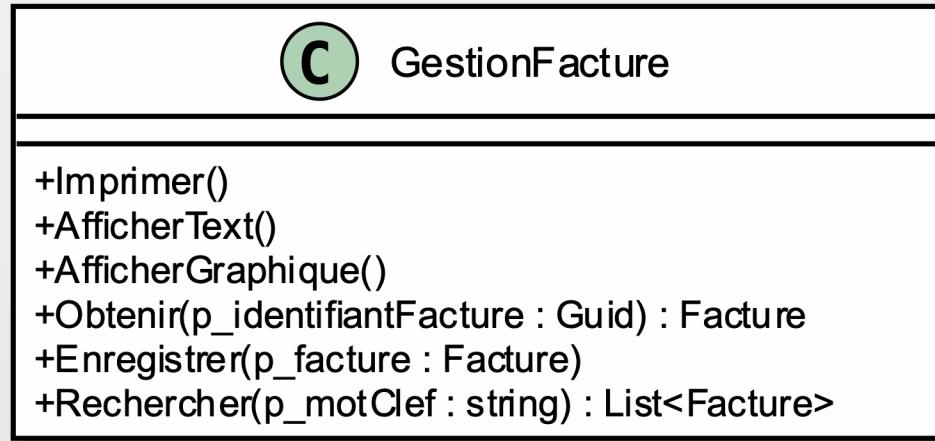
# *Interface segregation principle ISP*



**Interface Segregation Principle**  
You want me to plug this in *where*?

- = Ségrégation des interfaces
- Les interfaces doivent être spécifiques par domaine, donc il vaut mieux en implanter plusieurs qu'une seule générale
- Il ne faut pas dépendre d'actions dont on n'a pas besoin

# ISP – Violation

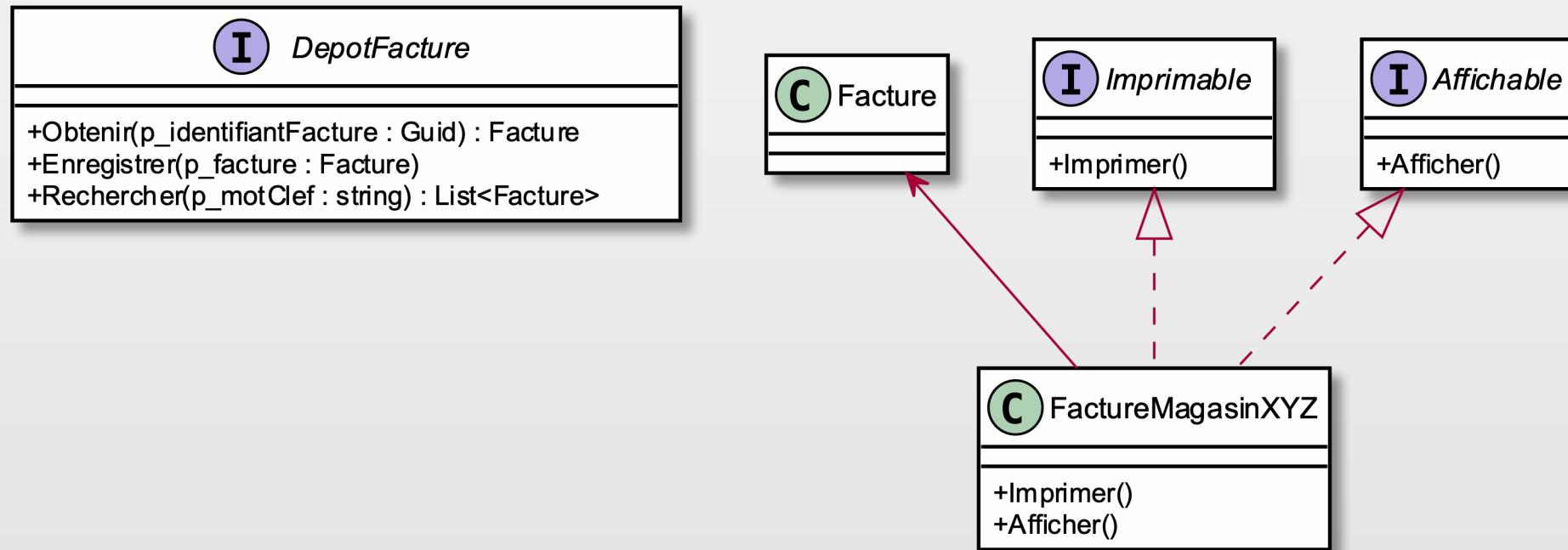


Attention dès que l'on voit Manager, Controller, Utils, etc. dans le nom de la classe, cela est généralement un signe de violation de l'ISP

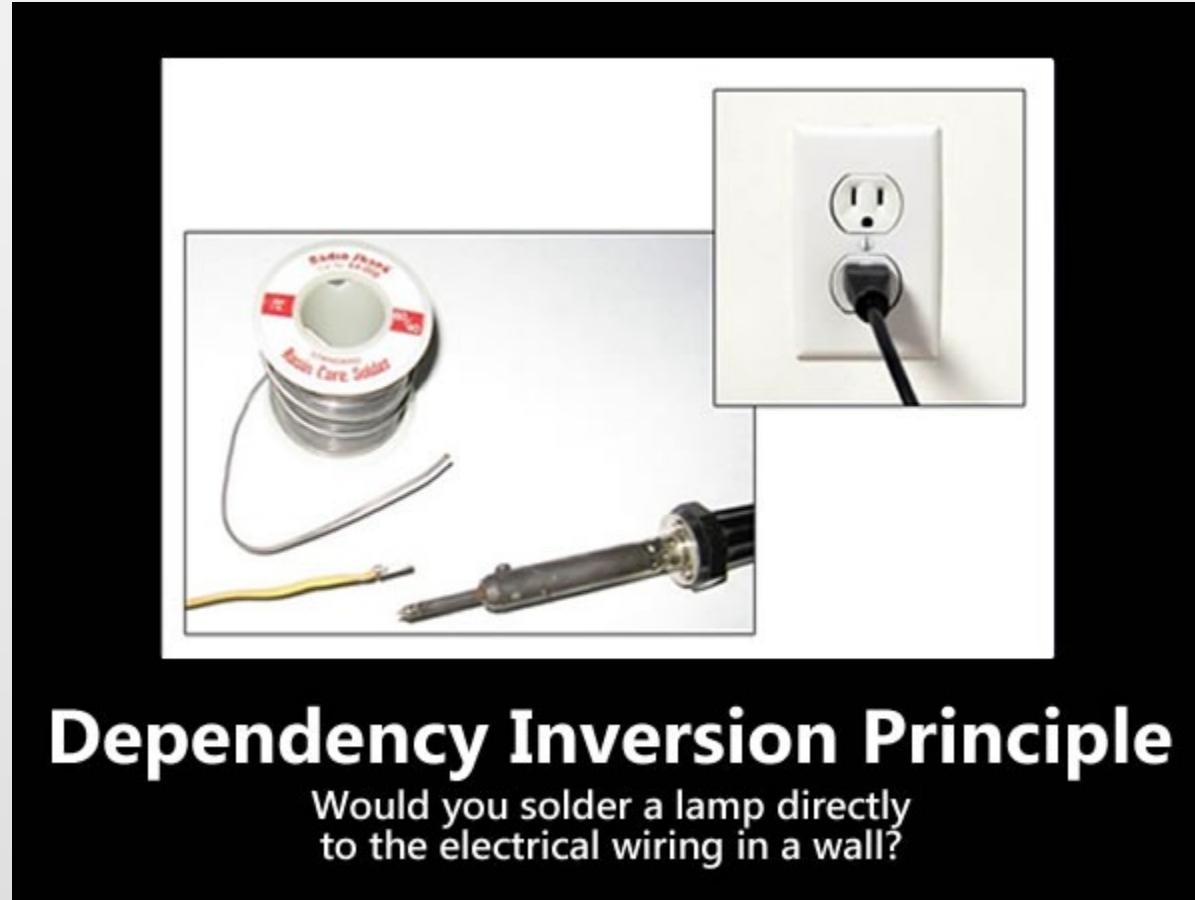
# ISP – exemple



Interface Segregation Principle  
You want me to plug this in *where*?



# *Dependency inversion principle – DIP*



# *Dependency inversion principle – DIP*



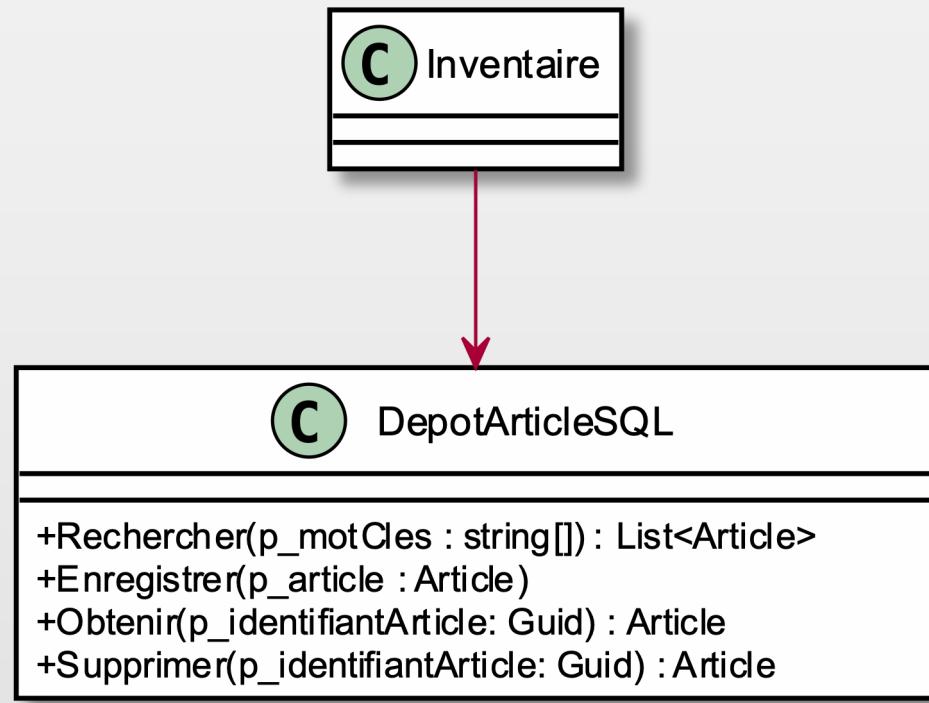
**Dependency Inversion Principle**  
Would you solder a lamp directly  
to the electrical wiring in a wall?

- = Inversion des dépendances
- La dépendance doit être par rapport aux interfaces et non sur les implantations
- Les modules de haut-niveau ne doivent pas dépendre de modules de bas-niveau
- Ces derniers doivent dépendre d'abstractions (ex. interfaces)

# DIP – Violation 1



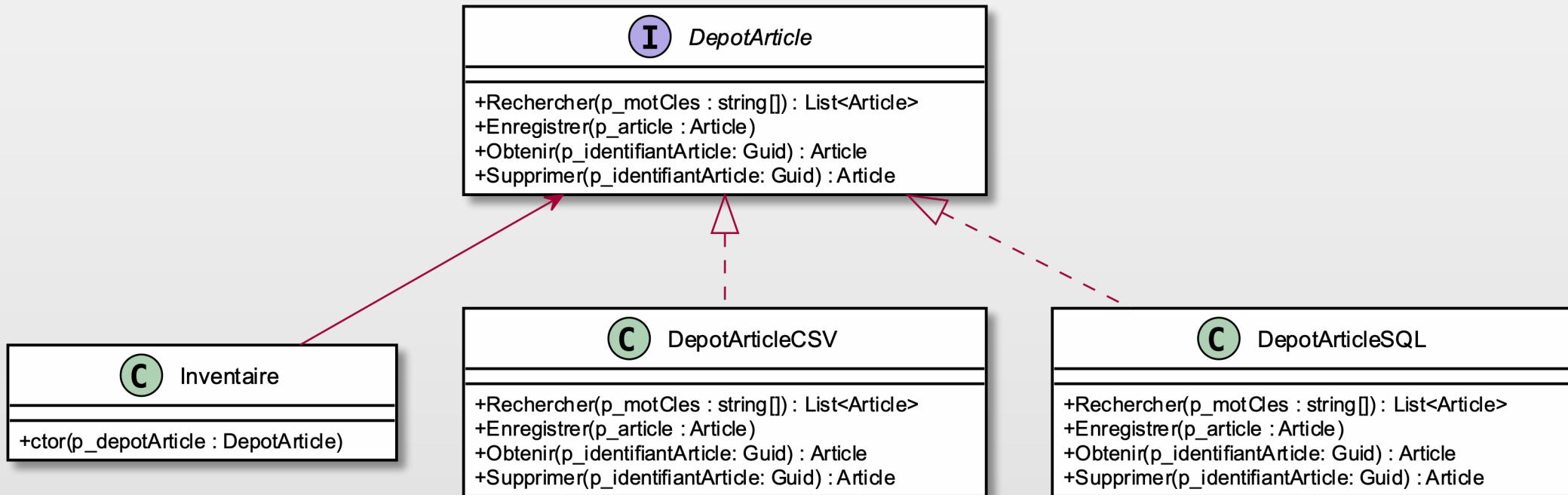
**Dependency Inversion Principle**  
Would you solder a lamp directly  
to the electrical wiring in a wall?



# DIP – Exemple résolution 1 – Pattern repository



**Dependency Inversion Principle**  
Would you solder a lamp directly  
to the electrical wiring in a wall?



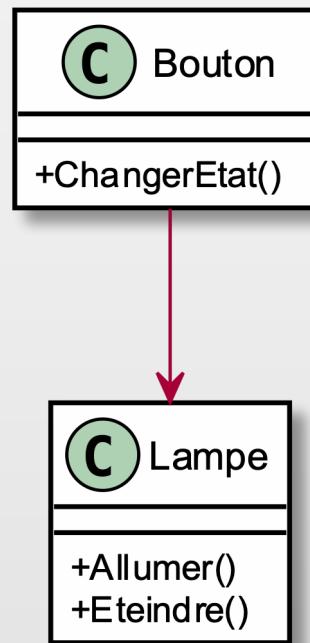
# DIP – Violation 2



**Dependency Inversion Principle**  
Would you solder a lamp directly  
to the electrical wiring in a wall?

Exercice :

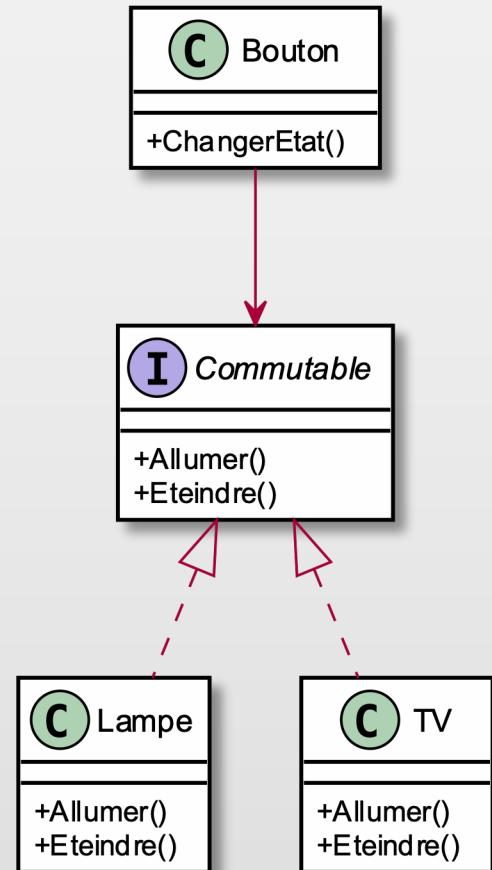
- Comment ajouter un nouvel appareil, par exemple une TV, qui est aussi allumable par un Bouton ?
- Modifiez le diagramme en conséquence



# DIP – Exemple résolution 2 – Abstract server



**Dependency Inversion Principle**  
Would you solder a lamp directly  
to the electrical wiring in a wall?



# *Oui mais tout ça pour quoi ? Conseils*

- Limiter les dépendances, donc diminuer le couplage
  - Augmenter la cohésion
- ⇒ Limiter les ras de marée en cas de changement
- Exposer le quoi et non le comment : principe d'encapsulation (cacher le comment)
  - La POO est orientée objet et non orientée données

# *Oui mais tout ça pour quoi ? Conseils*

- La traduction de l'héritage par « est un » est dangereuse
  - ⇒ Il faut aussi exclure l'héritage comme moyen de réutilisation
  - ⇒ Ne pas confondre avec « a un » (composition : une voiture a des roues)
- Préférer la composition à l'héritage
  - ⇒ Utiliser les abstractions telles que les interfaces pour inverser les dépendances
- Ne pas oublier : YAGNI / DRY
- Suite : Tell don't ask / loi de Demeter / KISS

# Références

- SRP : [https://drive.google.com/file/d/0ByOwmqah\\_nuGNHEtcU5OekdDMkk/view](https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view)
- OCP :  
<https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWfkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view>
- LSP :  
<https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view>
- ISP :  
<https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view>
- DIP :  
<https://drive.google.com/file/d/0BwhCYaYDn8EgMjdIMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRINTQ3ZGMz/view>
- Ou référence globale : <http://butunclebob.com/Articles.UncleBob.PrinciplesOfOOD>
- Livre : Agile principles, patterns and Practices (in C#), Robert C. Martin et Micah Martin