

Ecrire du code S.O.L.I.D



Isabelle BLASQUEZ
@iblasquez



SOLID

Software Development is not a Jenga game

Novembre 2017

Rappels des principes de Bases de l'Orienté Objet

Classe : Abstraction des caractéristiques communes à un ensemble d'objets.

Une classe permet de créer des **objets** qui communiquent entre eux par des **messages**.



Encapsulation : Protection de l'information contenue dans un objet.

Principe permettant de regrouper les *données* et les *routines* permettant de lire ou manipuler ces données.

En POO, le but de l'encapsulation est de masquer les **attributs** et certaines **méthodes** afin de ne rendre disponible que le comportement souhaité.

Héritage : Transmission des caractéristiques à ses descendants.

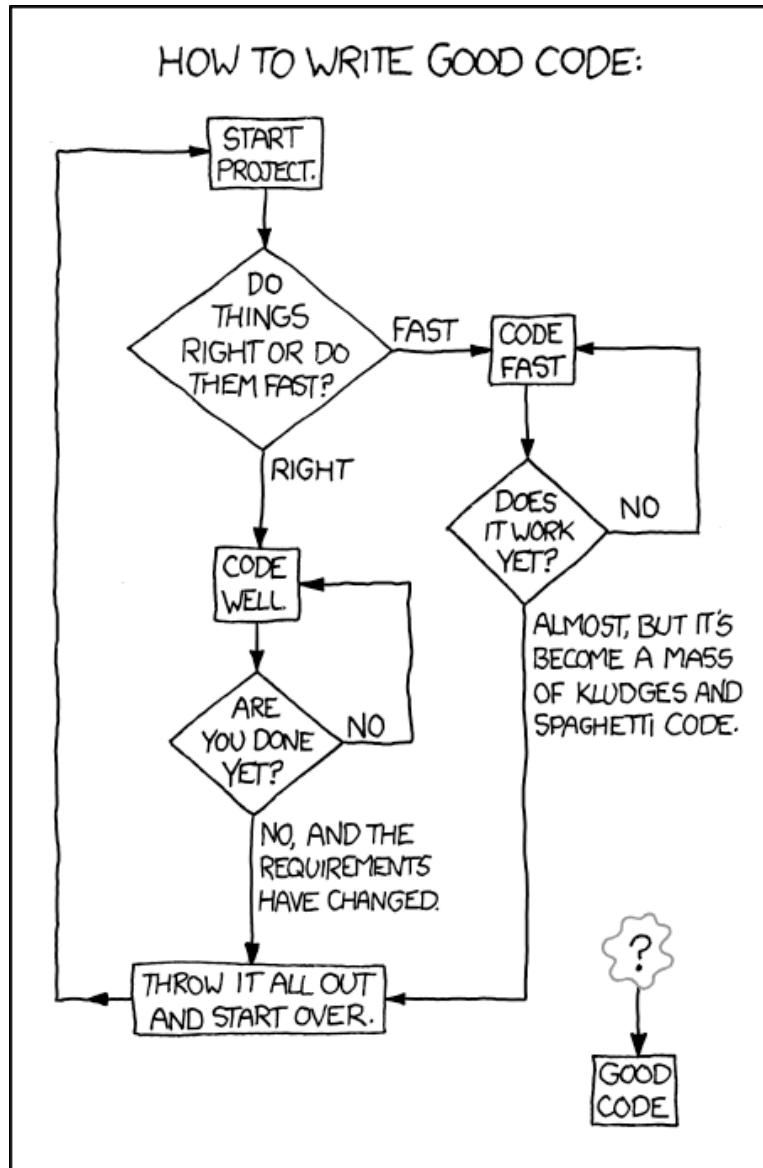
Mécanisme permettant, lors de la déclaration d'une nouvelle classe (fille), d'y inclure les caractéristiques d'une autre classe (mère).

Polymorphisme : du grec « qui peut prendre plusieurs formes »

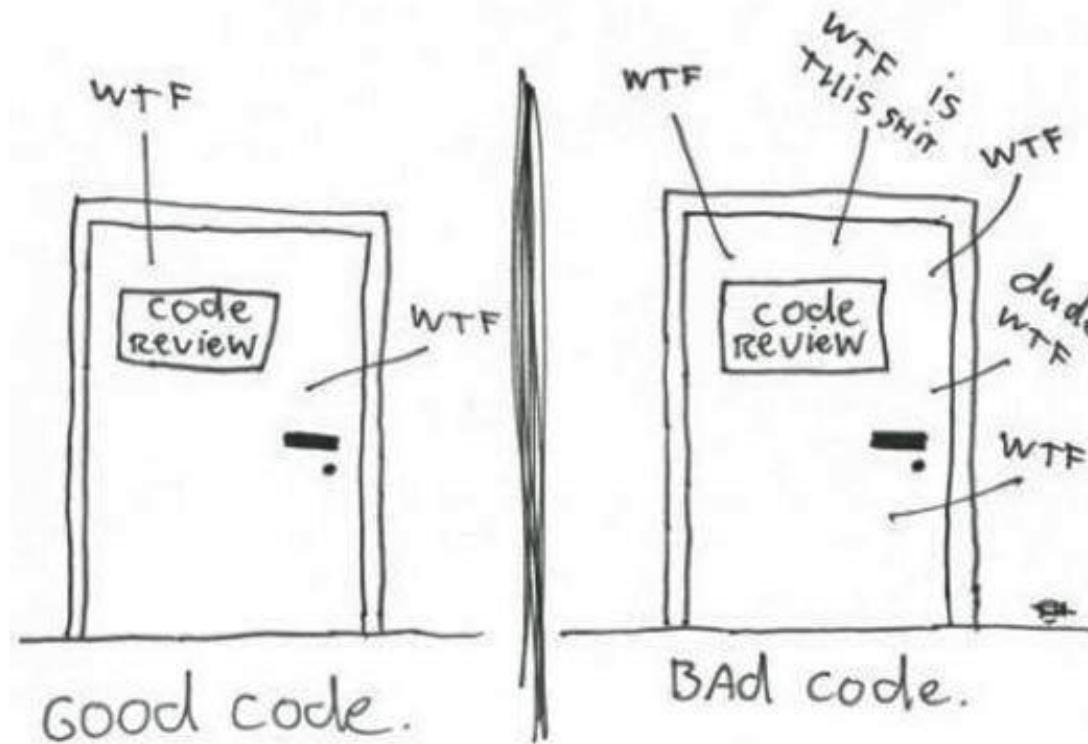
Concept consistant à fournir une interface unique à des entités pouvant avoir différents **types**.

- **polymorphisme d'héritage** (redéfinition, spécialisation ou *overriding*)
- **polymorphisme ad hoc** (surcharge ou *overloading*)
- **polymorphisme paramétrique** (généricité ou *template*)

Good code ?



The ONLY VALID MEASUREMENT
OF Code QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

La tolérance au changement est un des facteurs clés
du développement logiciel
(la maintenance demandant souvent des efforts croissants
au fil du temps et de l'évolution du logiciel)

Bad code ?... ou du moins 5 symptômes d'intolérance au changement

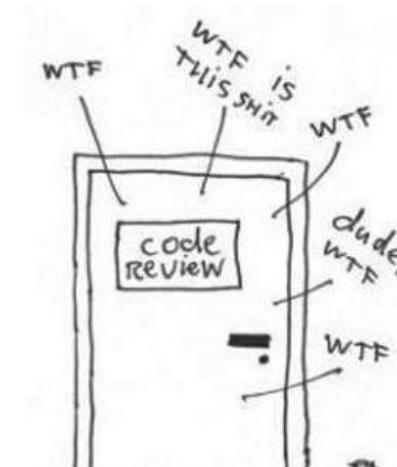
Rigidité: Chaque changement cause une cascade de modifications dans les modules dépendants (*pelote de laine*)



Fragilité: Tendance d'un logiciel à casser en plusieurs endroits à chaque modification. (= avec la rigidité : des erreurs dans des parties inattendues).)



Immobilisme: Incapacité d'un composant à pouvoir être réutilisé par d'autres projets ou par des parties de lui même.
(en raison d'un fort couplage : trop de dépendances)



Viscosité: Plus facile de faire un contournement plutôt que de respecter la conception qui a été pensée.
(plus facile de casser le design plutôt que de l'améliorer)

Opacité: par rapport à la lisibilité et simplicité de compréhension du code.

Extrait : <http://blog.xebia.fr/2011/07/18/les-principes-solid/>

A voir aussi (définitions en anglais) : http://www.fil.univ-lille1.fr/~routier/enseignement/licence/coo/cours/Principles_and_Patterns.pdf

et

<https://marcosantadev.com/solid-principles-applied-swift/>, <https://zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/> et

<https://www.youtube.com/watch?v=GtZtQ2VFweA>

Isabelle BLASQUEZ

COO & tolérance au changement

Les principes de base l'objet que sont
l'encapsulation, l'héritage et le polymorphisme
(voir M2103 et M2104)

ne suffisent pas seuls à guider correctement
la conception vers un logiciel tolérant au changement

*car la capacité d'**abstraction** n'est pas totalement intuitive
et raisonner de manière abstraite nécessite un certain recul*



L'utilisation de **principes de Conception Orientée Objet avancée**
va permettre d'engendrer un logiciel plus tolérant au changement c-a-d :

- **robuste** (où les changements n'introduisent pas de régression)
- **extensible** (où il est facile d'ajouter des fonctionnalités)
- et **réutilisable** (où les dépendances entre composants sont limitées)

Enoncés des Principes SOLID (principes de COO avancée)

Dans le livre [Agile Software Development, Principles, Patterns and Practices](#), Robert C. Martin a condensé, en 2002, 5 principes fondamentaux de conception, répondant à la problématique d'évolutivité , sous l'acronyme **SOLID** :

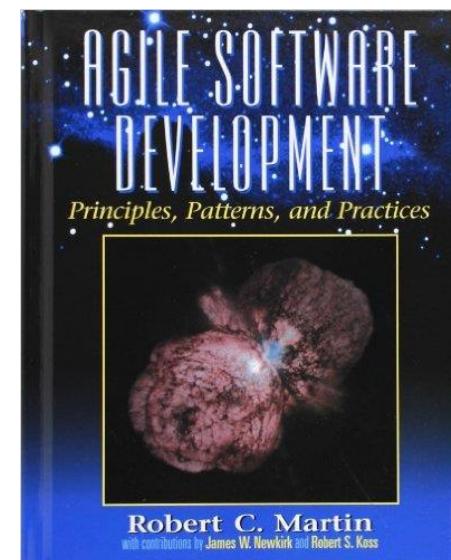
SRP : Single Responsibility Principle
(Principe de Responsabilité Unique)

OCP : Open-Closed Principle
(Principe d'ouverture/fermeture)

LSP : Liskov Substitution Principle
(Principe de substitution de Liskov)

ISP : Interface Segregation Principle
(Principe de Séparation des Interfaces)

DIP : Dependency Inversion Principle
(Principe d'inversion des dépendances)



Article de référence sur les principes SOLID le blog de Robert Martin (Uncle Bob)

<http://butunclebob.com/Articles.UncleBob.PrinciplesOfOOD>

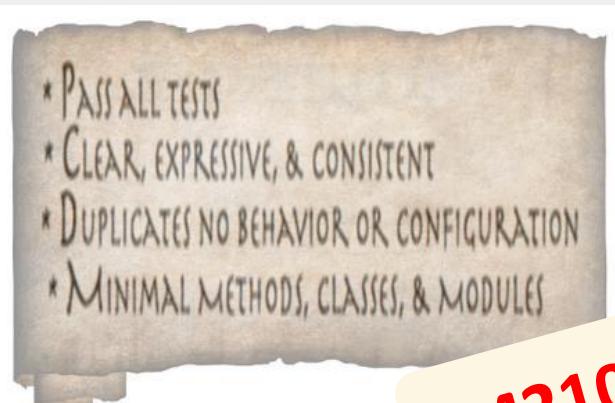
From STUPID code to good code (Simple & SOLID)

STUPID is an acronym that describes bad practices in Oriented Object Programming:

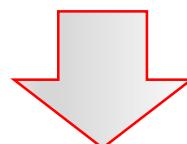
- Singleton
- Tight Coupling
- Untestability
- Premature Optimization
- Indescriptive Naming
- Duplication

Good Code ?

Simplicity Rules



M2104



*Nettoyer le code (**clean code**) et **refactorer***

SOLID is an acronym that stands for **five basic principles** of Object-Oriented Programming and design to fix STUPID code:



M3105

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

FROM STUPID TO SOLID CODE!

A few basic principles of Object-Oriented Programming and Design.

*... principes **SOLID** se retrouvant dans de nombreux **Design Patterns***

S.O.L.I.D

SRP



SINGLE RESPONSIBILITY PRINCIPLE
Ce n'est pas parce qu'on peut le faire
qu'il faut le faire

Principe de Responsabilité Unique

Les principes SOLID : SRP

Principe de Responsabilité Unique

***Une classe doit avoir
une et une seule responsabilité***



SINGLE RESPONSIBILITY PRINCIPLE

Ce n'est pas parce qu'on peut le faire
qu'il faut le faire

Autrement dit,
chaque objet de votre système ne doit avoir qu'une seule responsabilité
et tous les services de cet objet doivent s'efforcer d'assumer cette unique responsabilité

SRP dans la vie de tous les jours



Une seule chose à la fois

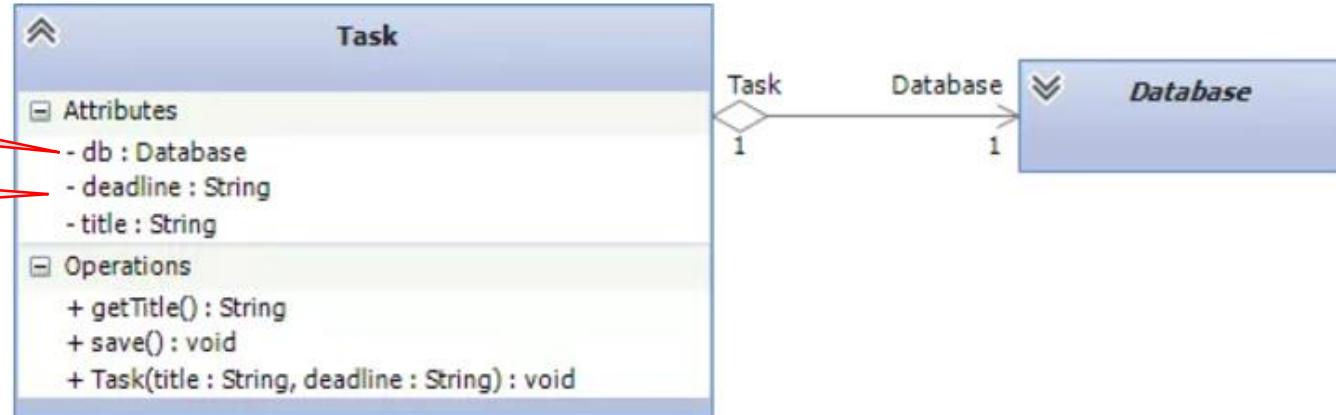
***Faire une seule chose
mais la faire bien***

***N'avoir qu'une seule raison
de changer***

Exemple de violation du principe SRP

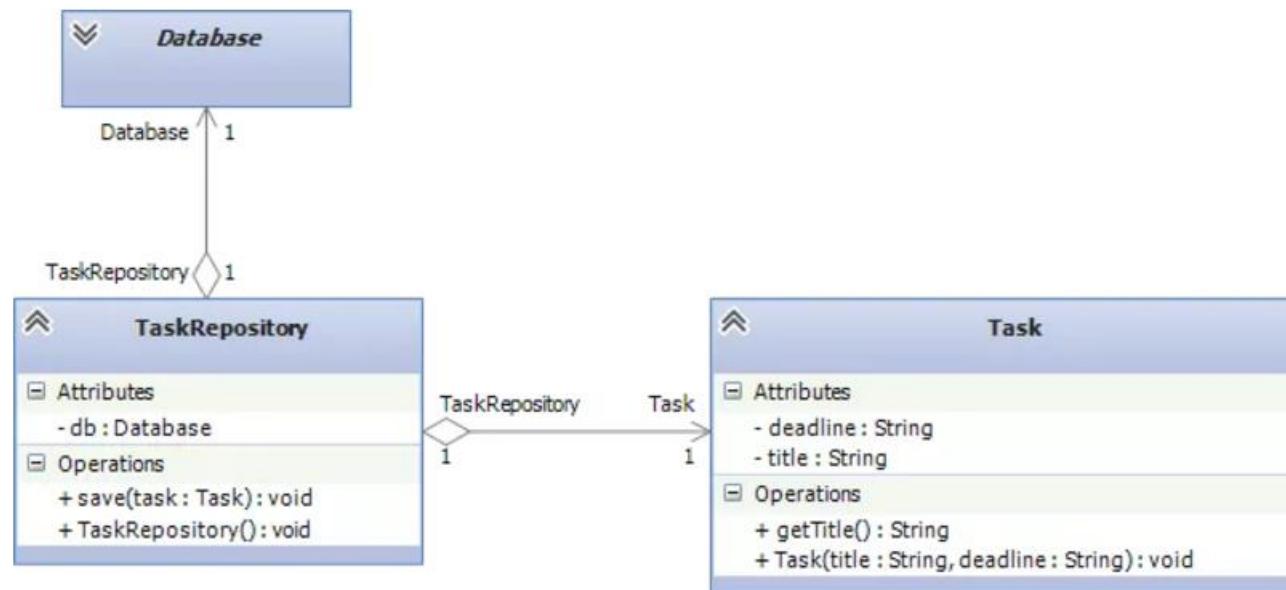
Accès aux données

Propriétés du modèle



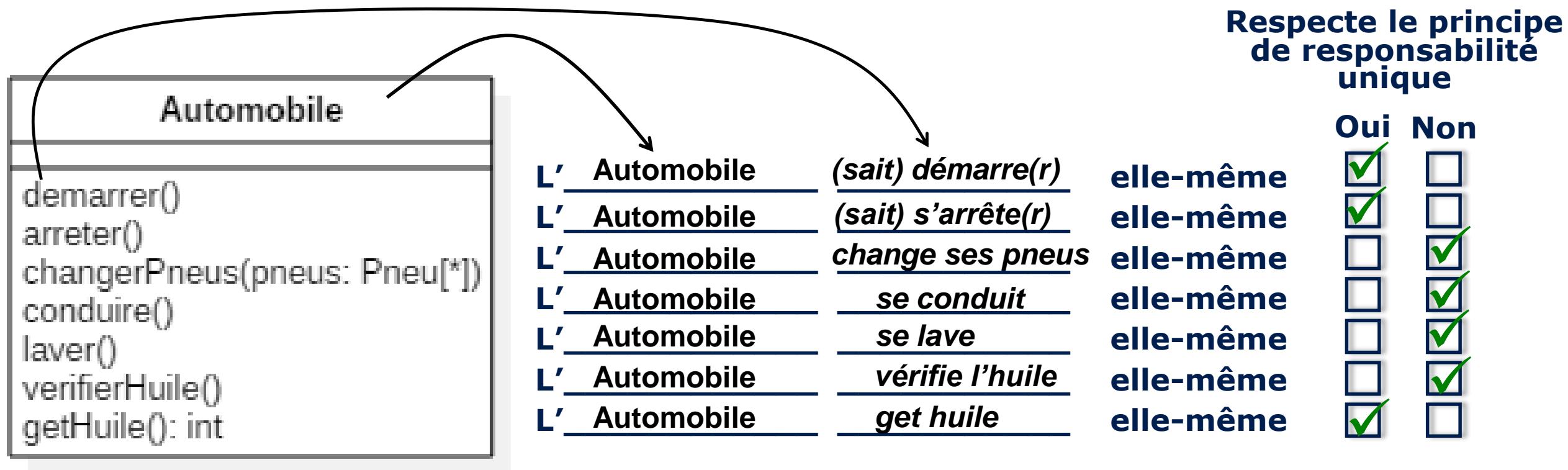
Violation du principe Single Responsabilité Principle \Rightarrow Refactoring en deux classes :

- Une classe responsable des propriétés du modèle (**Task**)
- Une classe responsable de l'accès aux données (**TaskRepository**)



Principe SRP : De responsabilités multiples à une responsabilité unique ... (1/2)

1. Analyser en repérant les responsabilités multiples



Principe SRP : De responsabilités multiples à une responsabilité unique ... (1'/2)

Possibilité d'utiliser une **carte CRC**

Classe, Responsabilités, Collaborateurs

Automobile
demarrer()
arreter()
changerPneus(pneus: Pneu[*])
conduire()
laver()
verifierHuile()
getHuile(): int

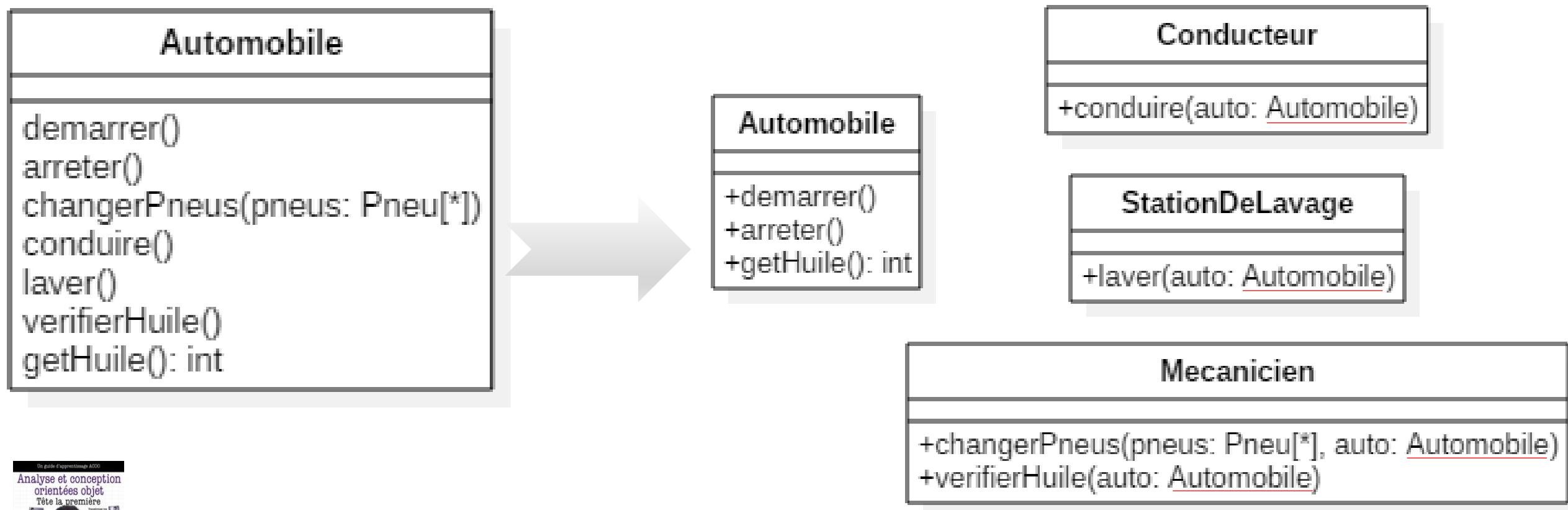
Classe : Automobile	
Description : Cette classe représente une voiture et ses fonctionnalités associées	
Responsabilités :	
Nom	(classes) Collaboratrices
<i>Démarre toute seule</i>	
<i>S'arrête toute seule</i>	
Se fait changer les pneus	Mécanicien, Pneu
Se fait conduire	Conducteur
Se fait laver	Station de Lavage, Employé
Se fait vérifier l'huile	Mécanicien
<i>Signale le niveau d'huile</i>	

« Se fait » : a priori, ce n'est pas une responsabilité de cette classe

pour comprendre qu'elle doit être la **responsabilité d'une classe** et avec quelles autres classes elle doit **collaborer**

Principe SRP : De responsabilités multiples à une responsabilité unique ... (2/2)

2. Refactorer en déplaçant chaque méthode dans une classe approprié à sa responsabilité



Vous avez implémenté correctement le principe de responsabilité unique si chacun de vos objets n'a qu'une seule raison de changer

How many responsibilities?

```
class Employee {  
    public Pay calculatePay() {...}  
    public void save() {...}  
    public String describeEmployee() {...}  
}
```

3 responsabilités distinctes
⇒ 3 classes à faire apparaître !!!

SRP en pratique et en vidéo !

[Crafties] Episode #2 - Single Responsibility Principle



Crafties

```
graph TD; CGA[Computational Geometry Application] --> R[Rectangle<br/>+ draw()  
+ area() : double]; R --> GUI[GUI]; GA[Graphical Application] --> R; GA --> GUI;
```

srp [C:/dev/crafties/srp] - [srp] - .../src/main/java/com/crafties/srp/Square.java - IntelliJ IDEA 15.0.1

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

srp src main java com crafties srp Square

Project Structure

srp (.idea)

src

main

java

com.crafties.srp

geometry

gui

test

Screen

Square

GUIApplication.java

```
package com.crafties.srp;
import com.crafties.srp.gui.Screen;

public class Square {
    private final int edgeLength;
    public Square(int edgeLength) { this.edgeLength = edgeLength; }
    public void drawOn(Screen screen) { screen.print("I am a square of " + edgeLength + " cm edge."); }
    public int area() { return edgeLength * edgeLength; }
}
```

GeometryApplication.java

SquareTest

.gitignore

pom.xml

srp.iml

External Libraries

Kata de refactoring vers SRP :
À visualiser sur : <https://www.youtube.com/watch?v=-mroBIWUBBM>

Code disponible sur : <https://github.com/nphumbert/crafties-srp>

Isabelle BLASQUEZ

S.O.L.I.D
OCP



OPEN CLOSED PRINCIPLE
Une opération à cœur ouvert n'est pas nécessaire lorsqu'on enfile un vêtement

Principe Ouvert/Fermé

Image : <http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

OCP décrit par Robert C Martin <https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWfkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view>
accessible depuis <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Isabelle BLASQUEZ

Les principes SOLID : OCP (Principe ouvert/fermé)



OPEN CLOSED PRINCIPLE

Une opération à cœur ouvert n'est pas nécessaire lorsqu'on enfile un vêtement

Wrong
Class ABC{
FindAreaOfSquare()
FindAreaOfTriangle()
}

Right
Abstract Class Shape
{
FindArea()
}

Une classe doit être ouverte aux extensions, mais fermée aux modifications.
(possibilité d'étendre le comportement sans modifier la classe)



Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

Images extraites de : <http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

En savoir plus : http://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf

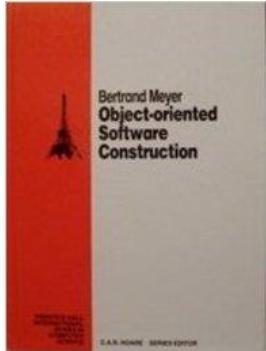
Extrait : <http://zereturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/>

<http://thetechnologistinyou.blogspot.fr/2012/09/solid-principles-quick-introduction.html>

Principe OCP

Principe énoncé par Bertrand Meyer

Les entités informatiques (paquetage, classe, méthode) doivent être ouvertes aux extensions mais fermées aux modifications.



Ouverture aux extensions :

Le comportement doit pouvoir être étendu

c-a-d **ouvrir** en autorisant la création de sous-classes pour ajouter de nouvelle(s) fonctionnalité(s), méthode(s).

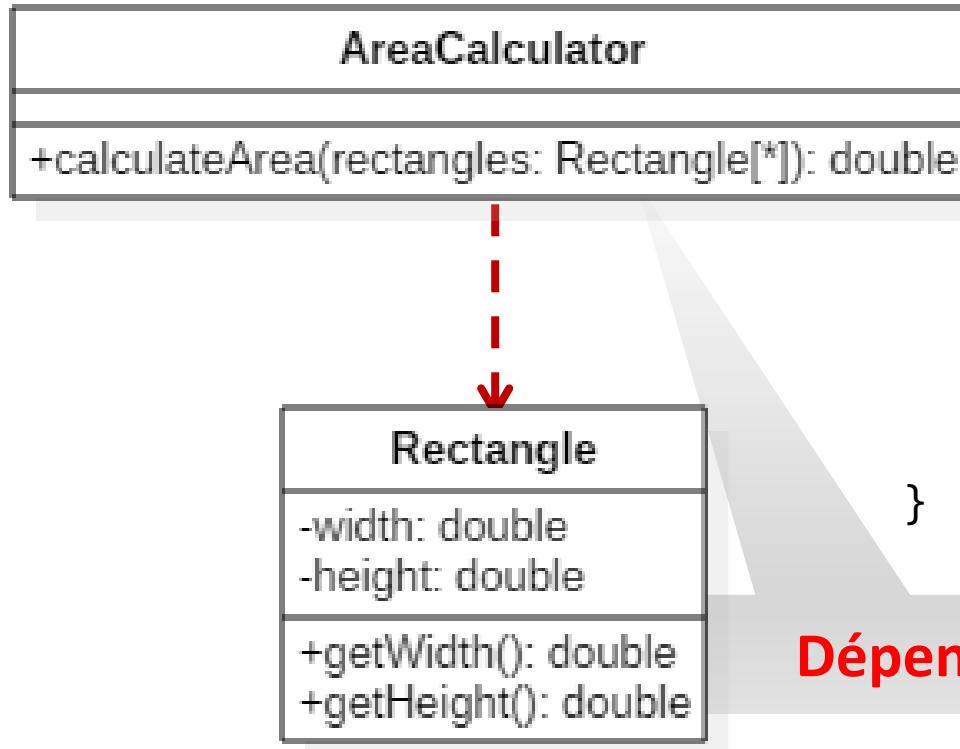
Fermeture aux modifications :

Le comportement doit pouvoir être étendu mais sans modification de son code source.

c-a-d **fermer** en n'autorisant personne à toucher à un code qui fonctionne...

En d'autres termes, l'ajout de fonctionnalités doit se faire
en ajoutant du code et non en éditant du code existant.

Illustration du principe OCP : calcul d'aire (1/3)

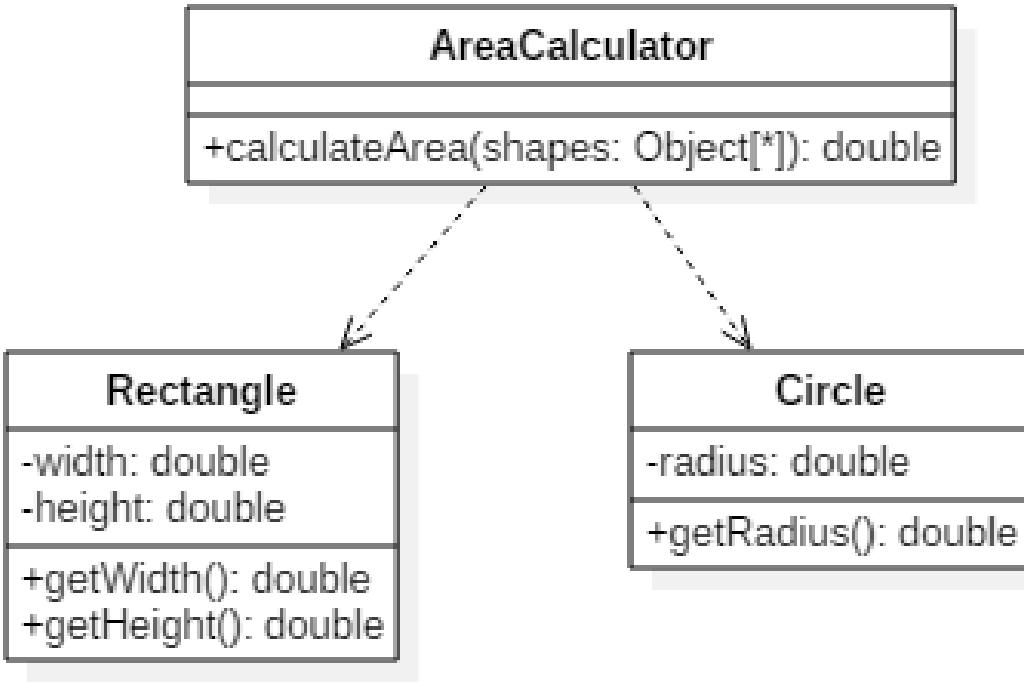


```
public class AreaCalculatorRectangle {  
  
    public double calculateArea(Rectangle[] rectangles) {  
        double result = 0;  
        for (Rectangle rectangle : rectangles) {  
            result += (rectangle.getWidth() * rectangle.getHeight());  
        }  
        return result;  
    }  
}
```

Dépendance

Illustration du principe OCP : calcul d'aire (2/3)

Extension du calcul d'aire à une nouvelle forme ...



```
public class AreaCalculator {  
  
    public double calculateArea(Object[] shapes) {  
        double area = 0.0;  
  
        for (Object shape : shapes) {  
  
            if (shape instanceof Rectangle) {  
                Rectangle rect = (Rectangle) shape;  
                area += (rect.getWidth() * rect.getHeight());  
            }  
  
            if (shape instanceof Circle) {  
                Circle circ = (Circle) shape;  
                area += Math.PI * (circ.getRadius() * circ.getRadius());  
            }  
        }  
        return area;  
    }  
}
```

A red callout box highlights the line `if (shape instanceof Triangle)`, indicating that this part of the code needs to be modified when adding a new shape type.

Extension à une nouvelle forme (Triangle)

⇒ modification du code de la classe **AreaCalculator** qui devrait restée fermée

⇒ **Violation du principe OCP !**

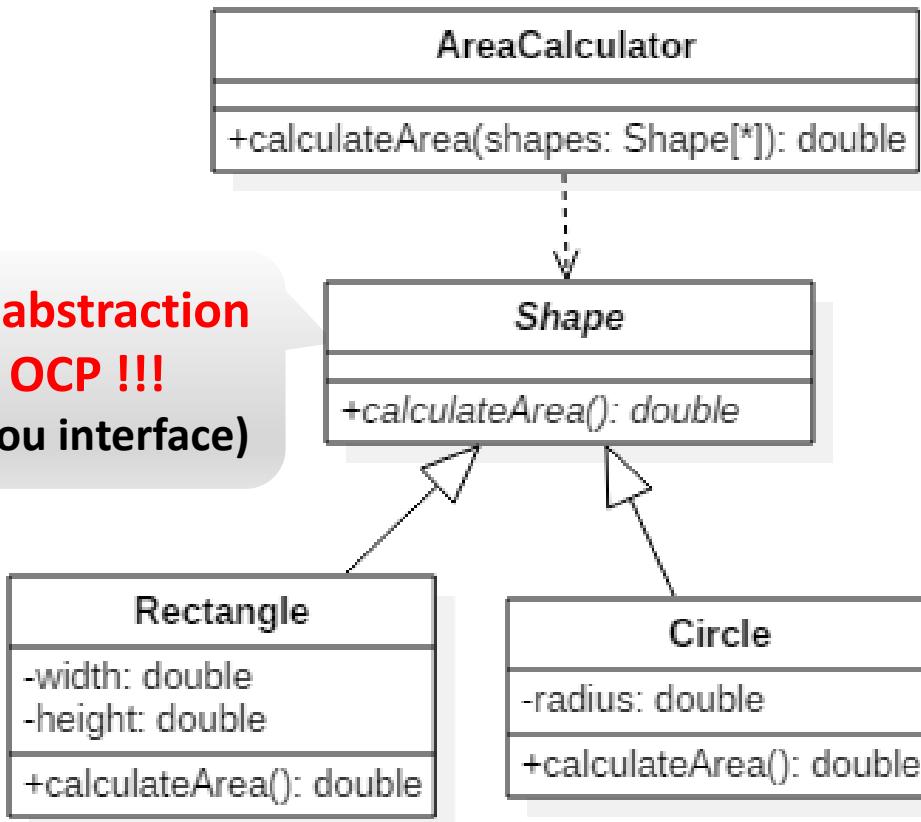
Illustration du principe OCP : calcul d'aire (3/3)

Respect OCP !

Utilisation d'une abstraction (`Shape`)

pour ouvrir la classe `AreaCalculator` et permettre l'extension de n'importe quelle forme

Passer par une abstraction
est la clé de OCP !!!
(classe abstraite ou interface)



```
public class AreaCalculator {

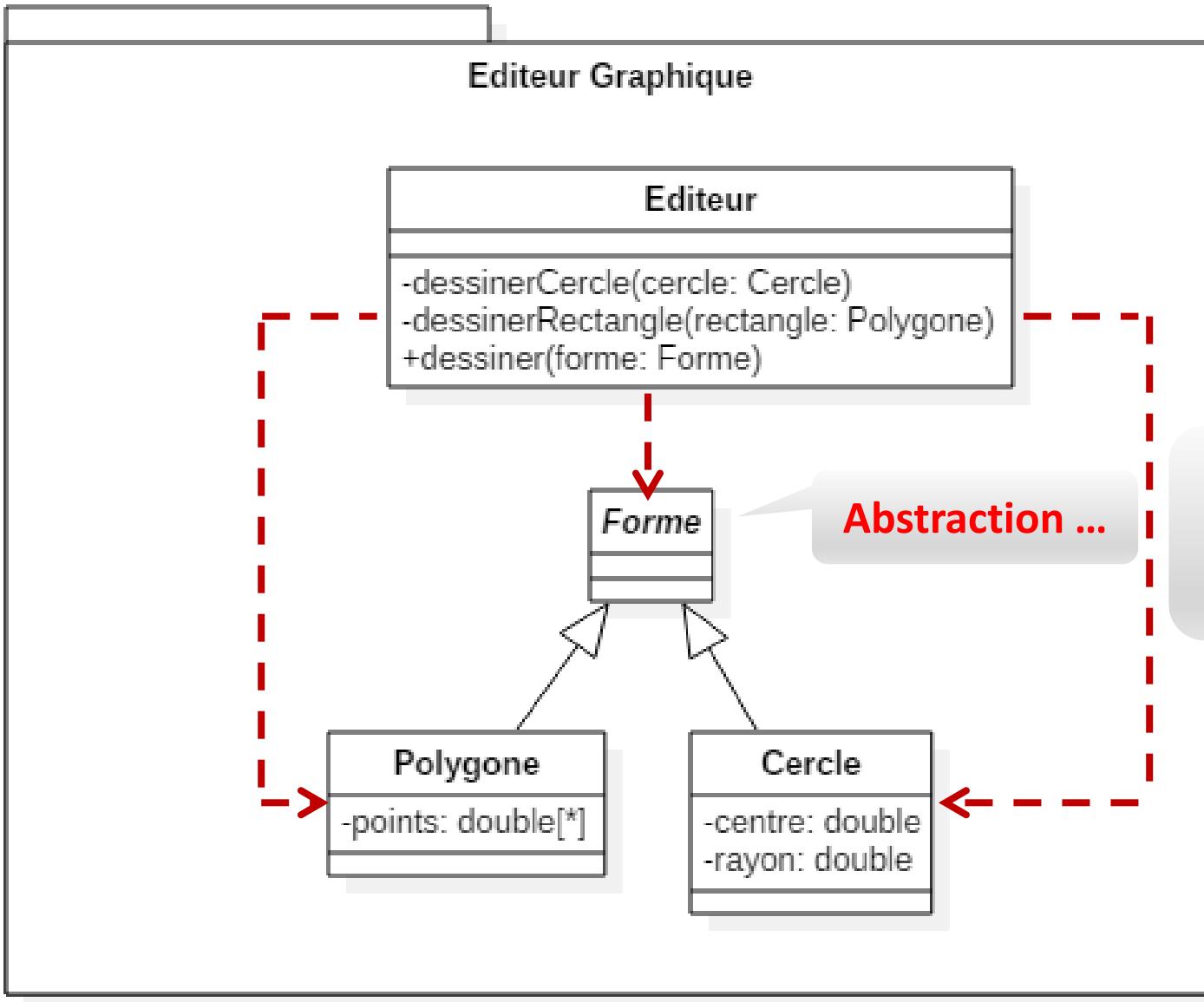
    public double calculateArea(Shape[] shapes) {
        double area = 0.0;
        for (Shape shape : shapes) {
            area += shape.calculateArea();
        }
        return area;
    }

    public class Circle extends Shape {
        int radius;

        public double calculateArea() {
            return Math.PI * this.radius * this.radius;
        }
    }

    public class Triangle extends Shape {
        //...
        public double calculateArea() { //...}
    }
}
```

Illustration du principe OCP : Editeur graphique (1/2)



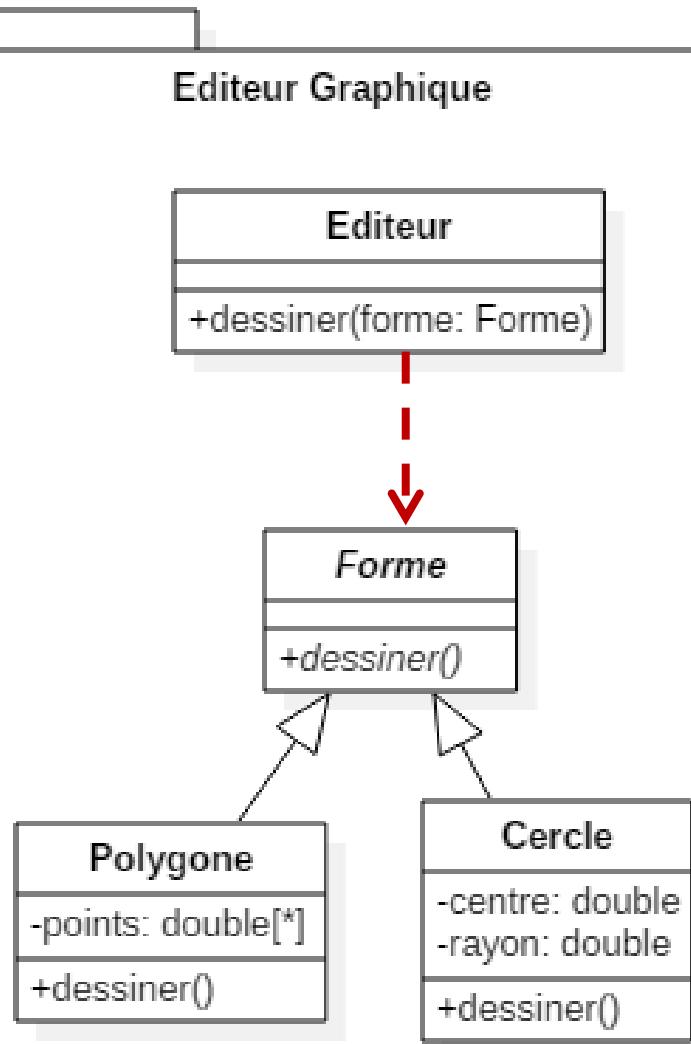
... mais mauvaise conception
car **Editeur** ouvert :

Trop de dépendances (une vers chaque classe)
une nouvelle **Forme** ⇒
une nouvelle méthode dans **Editeur**



Violation OCP !!!

Illustration du principe OCP : Editeur graphique (2/2)



Utiliser seulement l'**abstraction** ne suffit pas,
il faut aussi bien veiller à
limiter les dépendances
en faisant bon usage du **polymorphisme** !

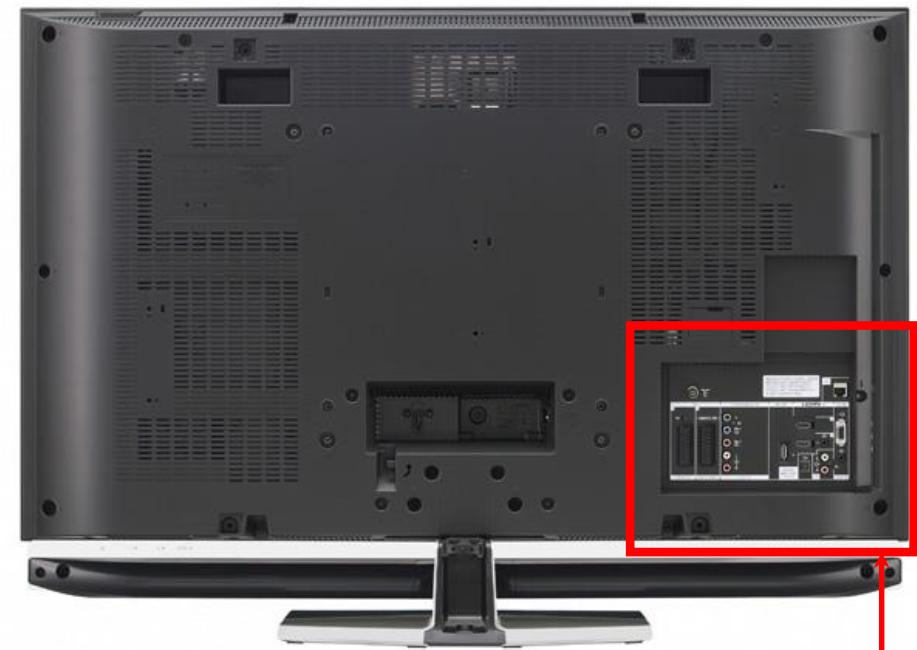
(`dessiner` est délégué à `Forme`
puis implémenté dans chaque classe concrète)

Respect OCP : Etendre le comportement sans modifier le code

(possibilité d'ajouter une classe Ellipse qui hérite de Forme et
implémente le nouveau comportement dans sa propre méthode dessiner).

OCP dans la vie de tous les jours

La connectivité d'une TV ne se résume plus depuis longtemps au câble péritel....



La télévision :
ouverte à l'extension,
mais fermée à la modification

Via **l'interface de connectique**,
on branche uniquement le(s) câbles dont
on a besoin SANS toucher au comportement de la TV
(« *sans ouvrir les entrailles de la TV et bidouiller ;-)* »)

Les design patterns et l'OCP

Un **design pattern (patron de conception)** présente **une bonne pratique** en réponse à un problème de conception récurrent. Il décrit une **solution standard**, qui adaptée au contexte, peut-être réutilisée dans la conception de différents logiciels.

Les design patterns mettent en œuvre les principes **SOLID**.

Exemples de design pattern du GoF mettant en œuvre l'OCP

→ **Patron de Méthode**

→ **Stratégie**

Un Exemple d'utilisation de ces deux patterns : <https://code.tutsplus.com/tutorials/solid-part-2-the-openclosed-principle--net-36600>

→ **Fabrique**

→ **Visiteur**

→ ...



OCP en pratique et en vidéo !

SCREENCAST SUR L'OPEN/CLOSED PRINCIPLE

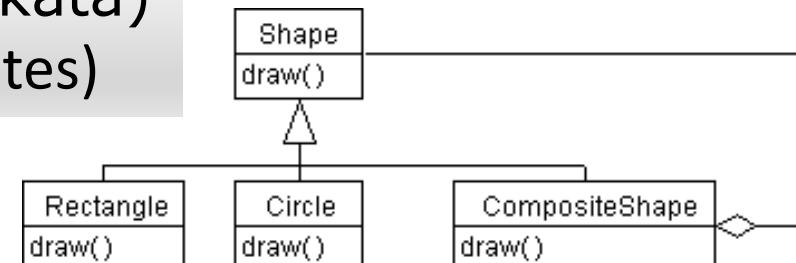
Avec cette vidéo, nous vous proposons d'étudier un des principes **SOLID** : l'open/closed principle. En résumé, ce principe permet simplement de garantir que le système que vous développez pourra facilement accueillir de nouvelles fonctionnalités (ouvert aux extensions) sans toutefois devoir modifier le code existant (fermé aux modifications). Le principe va être illustré dans ce screencast en prenant l'exemple d'un code qui mérite effectivement qu'on se penche sur son extensibilité.

Si vous voulez faire ce Kata vous même, les sources sont disponibles sur github. (<https://github.com/dlresende/kata-geometric-modeling-system>)

SOLID

Extrait : <http://blog.xebia.fr/2014/05/19/screencast-sur-lopenclosed-principle/>

Kata de refactoring OCP (Geometric modeling system kata)
À visualiser sur : <https://vimeo.com/95051907> (10 minutes)



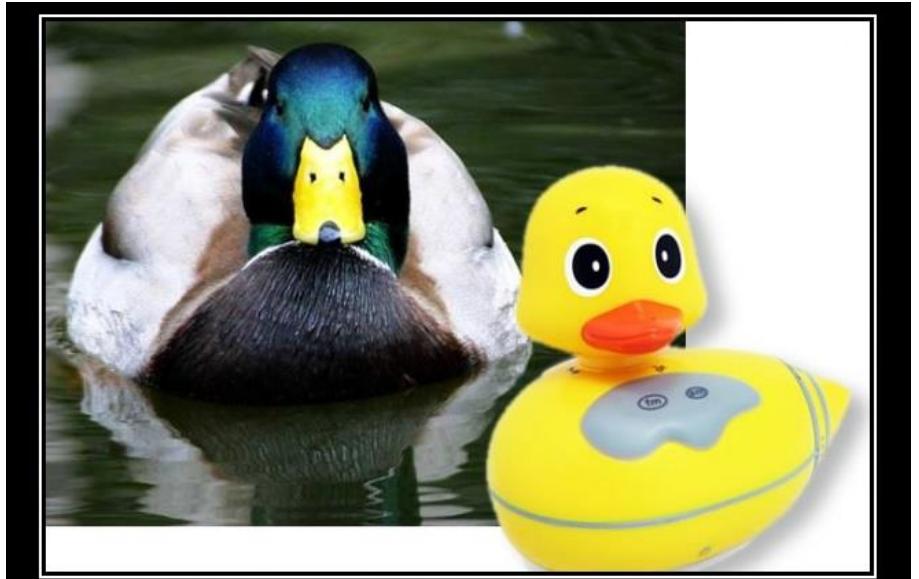
Dans cette vidéo, utilisation de la commande de remisage git : `git stash`

En savoir plus ... <https://git-scm.com/book/fr/v1/Utilitaires-Git-Le-remisage>

Isabelle BLASQUEZ

S.O.L.I.D

LSP



LISKOV SUBSTITUTION PRINCIPLE

Ca ressemble à un canard,
ça cancane comme un canard, mais à besoin de piles.
Vous avez surely la mauvaise abstraction.

Principe de Substitution de Liskov

Image : <http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

LSP décrit par Robert C Martin <https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view>
accessible depuis <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Isabelle BLASQUEZ

Les principes SOLID : LSP (Principe de substitution de Liskov)

Principe de Substitution énoncé par Barbara Liskov et Jeannette Wing (1993)

Si $q(x)$ est une propriété démontrable pour tout objet x de type T , alors $q(y)$ est vraie pour tout objet y de type S tel que S est un sous-type de T .

Autrement dit par Robert C. Martin ... Principe de Substitution en COO :

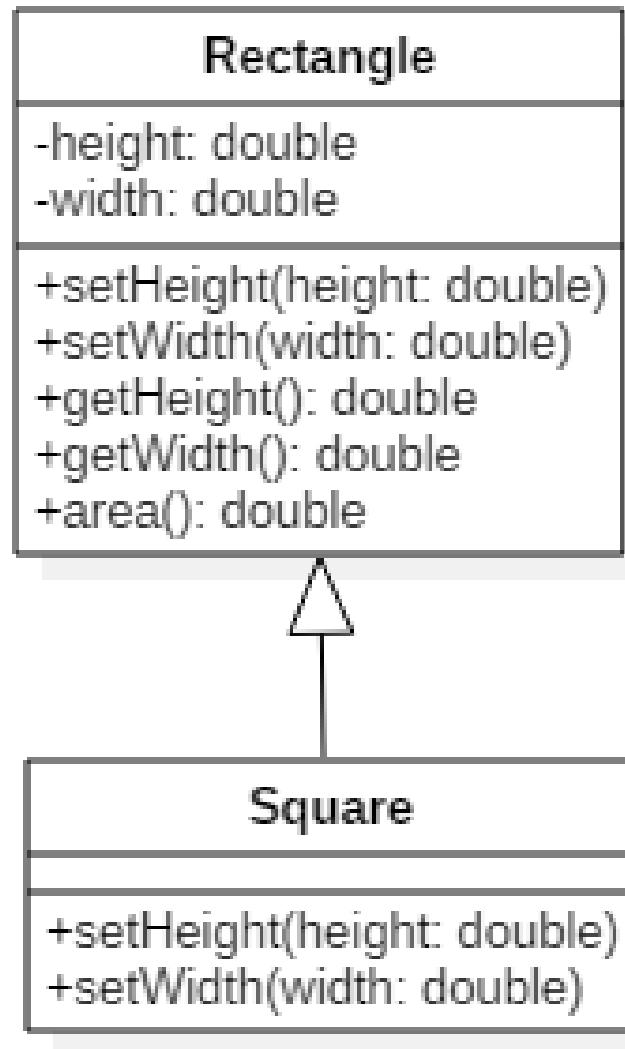
Les sous-types doivent pouvoir être substitués à leurs types de base.

c-a-d que les sous-classes doivent pouvoir remplacer leur classe de base et que les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser "inconsciemment" des objets dérivés de cette classe

Le **principe de substitution de Liskov** est une définition particulière de la **notion de sous-type**. Il permet de s'interroger sur la **bonne conception de l'héritage**

Illustration du principe LSP: un héritage bien nécessaire? (1/3)

Mathématiquement parlant,
un carré est un rectangle particulier.



```
public class Rectangle {
    protected double width;
    protected double height;

    public double getWidth() { return this.width; }

    public double getHeight() { return this.height; }

    public void setWidth(double width) {this.width = width; }

    public void setHeight(double height) {this.height = height; }

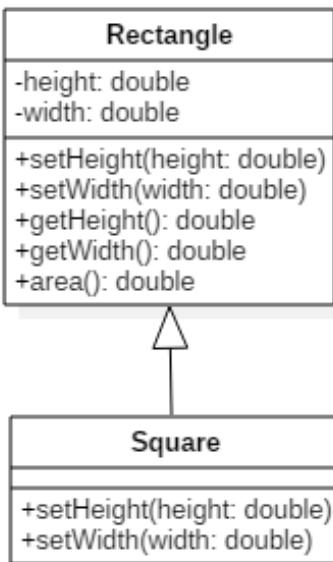
    public double area () {return this.width* this.height; }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(double width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

... Et pourtant ...

Illustration du principe LSP: un héritage bien nécessaire? (2/3)



```
public class LiskovTest {

    @Test
    public void shouldComputeArea() {
        Rectangle rectangle = new Square();
        rectangle.setWidth(5.0);
        rectangle.setHeight(10.0);
        assertEquals(50.0, rectangle.area(), 0.1);
    }
}
```

Compilation OK

MAIS ... Violation du « Contrat » à l'exécution :

Aire attendue : 50.0

Aire calculée : 100.0

A quoi sert un attribut hauteur dans un carré (`height`) ?

Runs: 1/1 Errors: 0 Failures: 1

AssertionError (expected:<50.0> but was:<100.0>) in LiskovTest.shouldComputeArea

Violation du principe LSP avec un tel héritage

Un carré ne peut pas se substituer à rectangle :

Le sous-type Square ne peut pas remplacer le type de base Rectangle !

Illustration du principe LSP: un héritage bien nécessaire? ... une solution alternative (3/3)

Nécessité d'un héritage ?

Bien que mathématiquement parlant un carré soit un rectangle,

du point de vue du comportement objet
un Square n'est pas un Rectangle.

Et le développement logiciel
se focalise sur le **comportement**

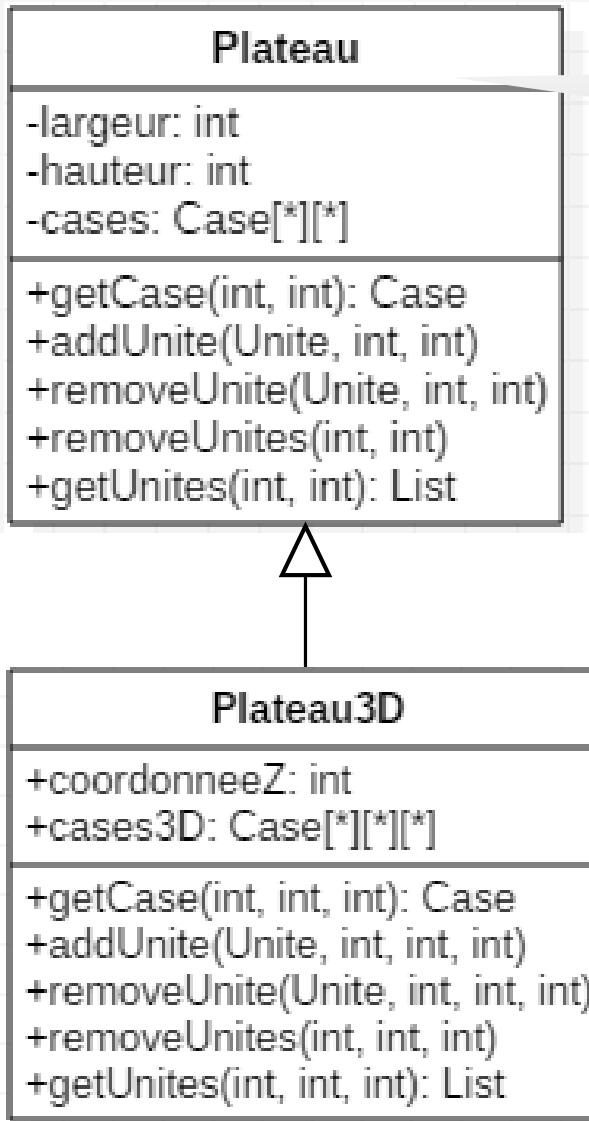
alors est-il vraiment pertinent de définir une classe Square ?

... Ne serait-il pas suffisant d'implémenter les méthodes suivantes dans la classe Rectangle pour pouvoir manipuler des carrés ?

```
public class Rectangle {  
    protected double width;  
    protected double height;  
  
    //...  
  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    public static Rectangle createSquare(int side) {  
        return new Rectangle(side, side);  
    }  
  
    public boolean isSquare() {  
        return getWidth() == getHeight();  
    }  
}
```



Illustration du principe LSP : un héritage judicieux ? (1/3)



Plateau conçu pour la création de jeux de stratégies autour de batailles terrestres (2D) entre unités de combat

Et si en plus des batailles terrestres,
le système de jeu permettait de créer des **batailles aériennes** ?

... Proposition d'extension du logiciel avec la classe **Plateau3D**
qui ajoute de nouvelles méthodes pour prendre des coordonnées 3D
Ce choix de conception est-il judicieux ?



Illustration du principe LSP : un héritage judicieux ? (2/3)

Plateau
-largeur: int
-hauteur: int
-cases: Case[*][*]
+getCase(int, int): Case
+addUnite(Unite, int, int)
+removeUnite(Unite, int, int)
+removeUnites(int, int)
+getUnites(int, int): List



Avec une relation de généralisation,
on hérite de tout !

Plateau3D
+coordonneeZ: int
+cases3D: Case[*][*][*]
+getCase(int, int, int): Case
+addUnite(Unite, int, int, int)
+removeUnite(Unite, int, int, int)
+removeUnites(int, int, int)
+getUnites(int, int, int): List

Plateau3D
-largeur: int
-hauteur: int
-coordonneeZ: int
-cases: Case[*][*]
-cases3D: Case[*][*][*]
+getCase(int, int): Case ???
+getCase(int, int, int): Case ???
+addUnite(Unite, int, int) ???
+addUnite(Unite, int, int, int)
+removeUnite(Unite, int, int) ???
+removeUnivite(Unite, int, int, int)
+removeUnites(int, int) ???
+removeUnites(int, int, int)
+getUnites(int, int): List ???
+getUnites(int, int, int): List

Plateau plateau = new Plateau3D();

Correct du point de vue de la compilation

Unite unite = plateau.getUnites(8,4);

... mais que signifie cette méthode
dans un plateau3D ?

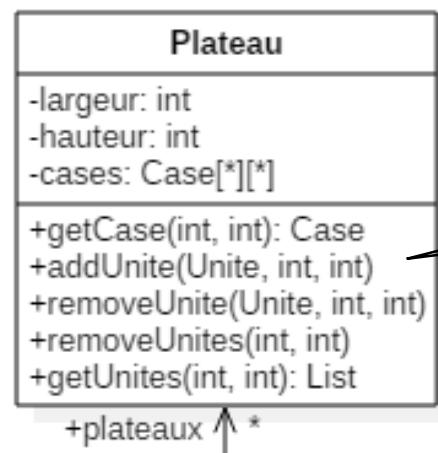
Difficile de comprendre un code qui utilise mal l'héritage

Violation du principe LSP

La classe Plateau3D ne peut être substituée à la classe Plateau
puisque aucune des méthodes de Plateau ne fonctionnent correctement dans un environnement 3D

Illustration du principe LSP : un héritage judicieux ? ... Héritage vs délégation (3/3)

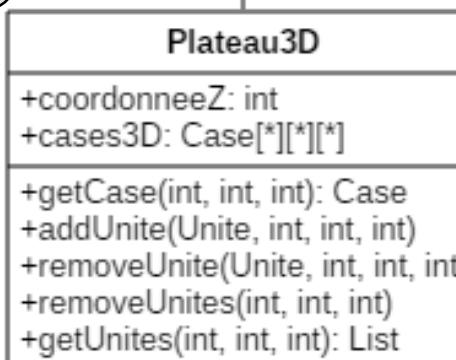
La délégation est la situation où vous confiez la responsabilité d'une tâche précise à une autre classe ou à une autre méthode



Plateau3D A-UN
un ensemble de plateaux

La classe Plateau a des fonctionnalités dont la classe Plateau3D a besoin, mais ce n'est pas le type de base pour Plateau3D

Plutôt qu'une extension (EST-UN), une **association** (A-UN) permet d'utiliser le comportement de Plateau sans enfreindre le principe LSP



Les méthodes de **Plateau3D** utilisent la `coordonneeZ` pour trouver quelle instance de **Plateau** utiliser, puis elles **délèguent** les coordonnées (x,y) aux fonctions de ce plateau : Elles **utilisent la fonctionnalité de Plateau** plutôt que de l'**étendre en la surchargeant**

Principe LSP : L'héritage n'est qu'une option ...

LSP s'assure que vous utilisez correctement l'héritage

EST-UN vs A-UN

Si vous trouvez du code qui enfreint ce principe, considérez la **délégation**, la **composition** ou **l'agrégation** pour rendre votre code plus souple, plus facile à maintenir, à étendre et à réutiliser

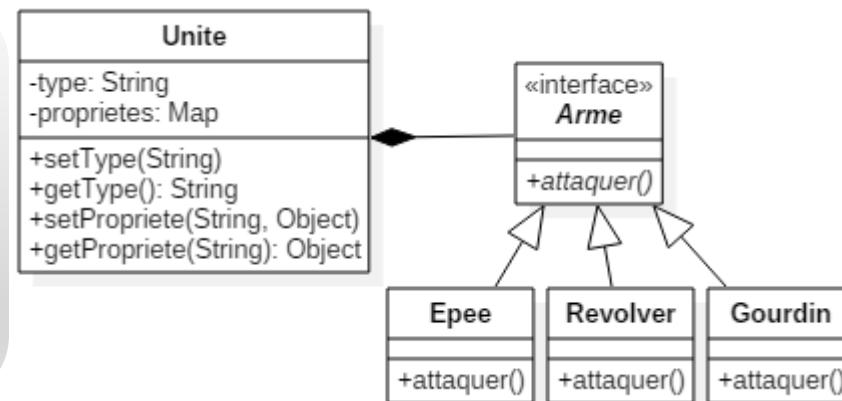
Délégation (association navigable)

Déléguéz un comportement à une autre classe quand vous ne voulez pas changer le comportement, mais que ce n'est pas de la responsabilité de votre objet d'implémenter ce comportement lui-même.

Composition

Permet d'utiliser le comportement d'une famille d'autres classes (souvent représentée par plusieurs implémentations d'une même interface) et de changer ce comportement pendant le fonctionnement de l'application.

Quand l'objet est détruit, tous ses comportements le sont aussi : les comportements n'existent pas en dehors de la composition elle-même.



Agrégation

Permet d'utiliser les comportements d'une autre classe sans limiter la durée de vie de ces comportements. Les comportements agrégés continuent à exister même après que l'objet qui les agrège ait été détruit.

Principes OO : une *bonne* conception réduit les risques ...

Encapsulez ce qui est varie

Chaque classe ne doit avoir qu'une seule raison de changer

Codez avec une interface plutôt qu'une implémentation

Les classes doivent se concentrer sur le comportement et la fonctionnalité

Tout objet de votre système ne doit avoir qu'une seule responsabilité et tous les services de l'objet doivent être orientés vers l'accomplissement de cette responsabilité (principe **SRP**)

Les classes doivent être ouvertes à l'extension mais fermées à la modification (principe **OCP**)

Les sous-classes doivent pouvoir être substitués à leur classe de base (principe **LSP**)

Evitez le code dupliqué en enlevant les choses communes, en les rendant abstraites et en les plaçant dans un endroit unique (principe de **non duplication**)



Inspiré de Analyse et Conception Orienté Objet – Tête la première

Isabelle BLASQUEZ

S.O.L.I.D

ISP



INTERFACE SEGREGATION PRINCIPLE
Où voulez-vous brancher cela ?

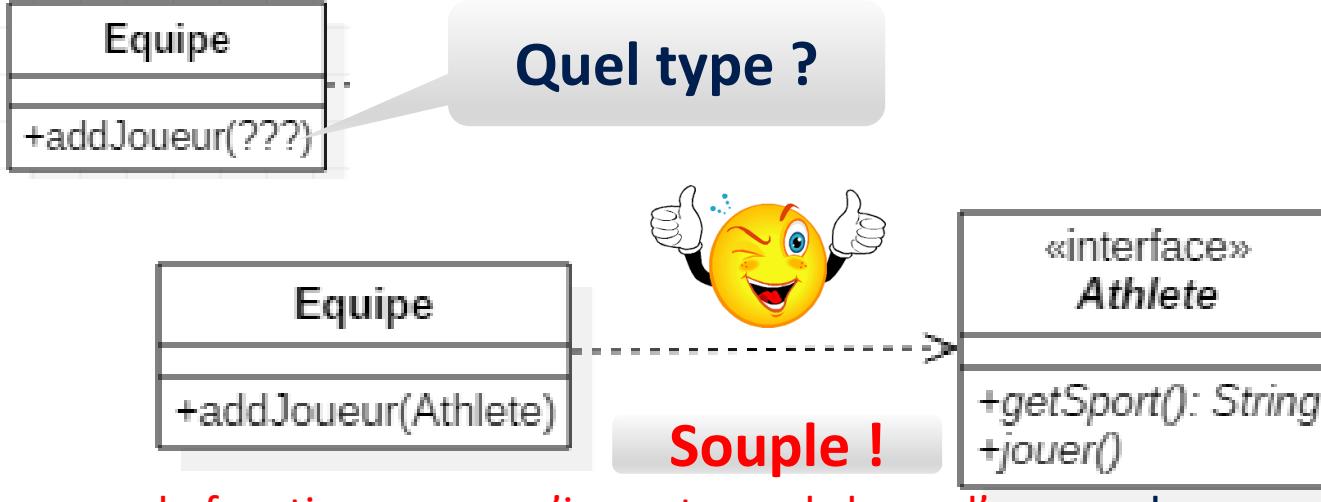
Principe de Ségrégation des Interfaces

Image : <http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

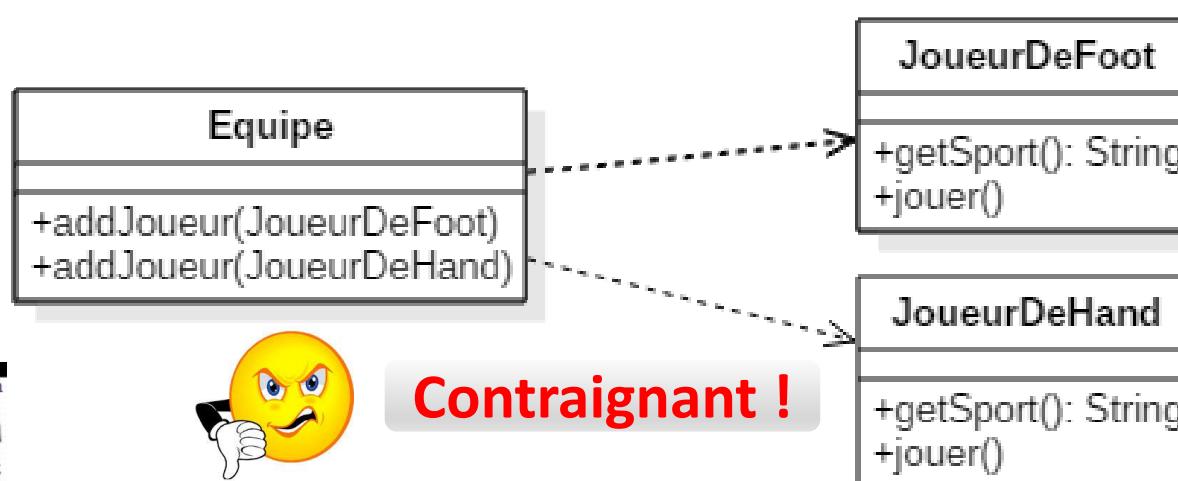
LSP décrit par Robert C Martin <https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view>
accessible depuis <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Isabelle BLASQUEZ

Retour sur un *bon* principe OO : Coder avec une interface plutôt qu'une implémentation c-a-d utiliser les interfaces comme des types



car code fonctionne avec n'importe quel classe d'Athlète !



Coder avec une **interface** permet de réduire les dépendances, rendant ainsi votre logiciel plus facile à étendre...



Les principes SOLID : ISP (Principe de ségrégation des Interfaces)



INTERFACE SEGREGATION PRINCIPLE
Où voulez-vous brancher cela ?

c-a-d préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale ...

... Mais



Un client ne devrait jamais être forcé de dépendre d'une interface qu'il n'utilise pas.



Interface Segregation Principle

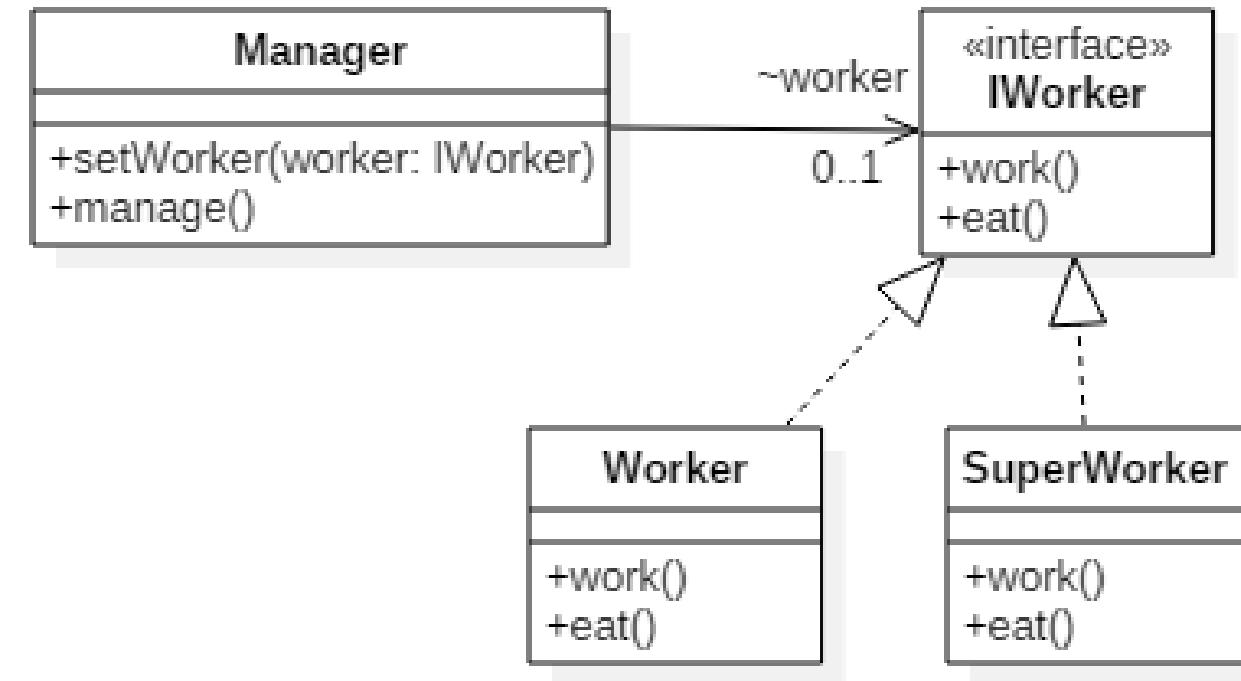
If IRequireFood, I want to Eat(Food food) not, LightCandelabra() or LayoutCutlery(CutleryLayout preferredLayout)

Isabelle BLASQUEZ

Illustration du principe ISP: Une interface commune ?

(1/4)

Des employés partagent une interface commune permettant de travailler (**work**) et de se restaurer (**eat**) ...



```
public class Worker implements IWorker{
    public void work() {
        // ....working
    }

    public void eat() {
        // ..... eating in launch break
    }
}
```

```
public class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker=w;
    }

    public void manage() {
        worker.work();
    }
}
```

```
public interface IWorker {
    public void work();
    public void eat();
}
```

```
public class SuperWorker implements IWorker{
    public void work() {
        //..... working much more
    }

    public void eat() {
        //.... eating in launch break
    }
}
```

Illustration du principe ISP: Une interface commune ?

(2/4)

... mais la technologie évoluant, on fait intervenir des travailleurs robots,
et ces derniers n'ont pas besoin de manger.

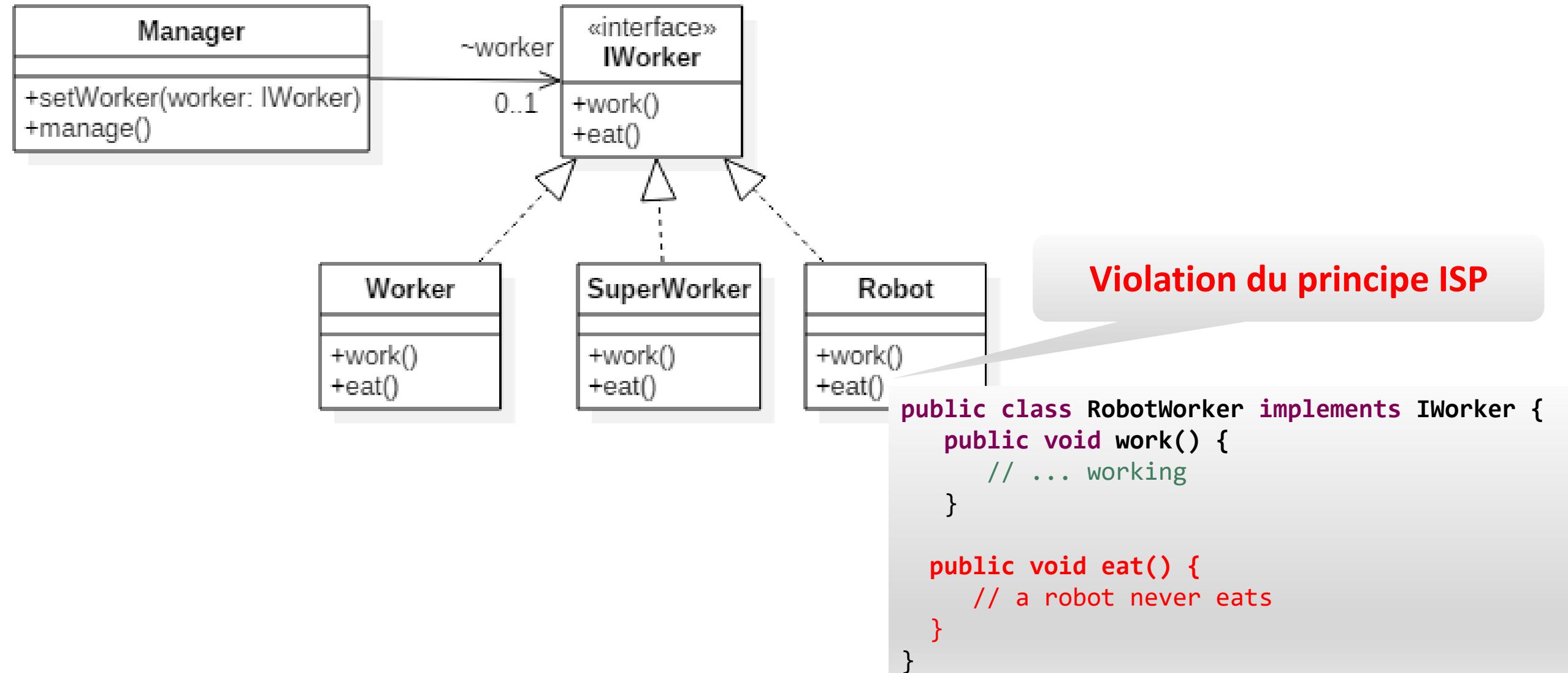


Illustration du principe ISP: Une interface commune ? NON !!! Séparons (3/4)

Pour respecter ISP

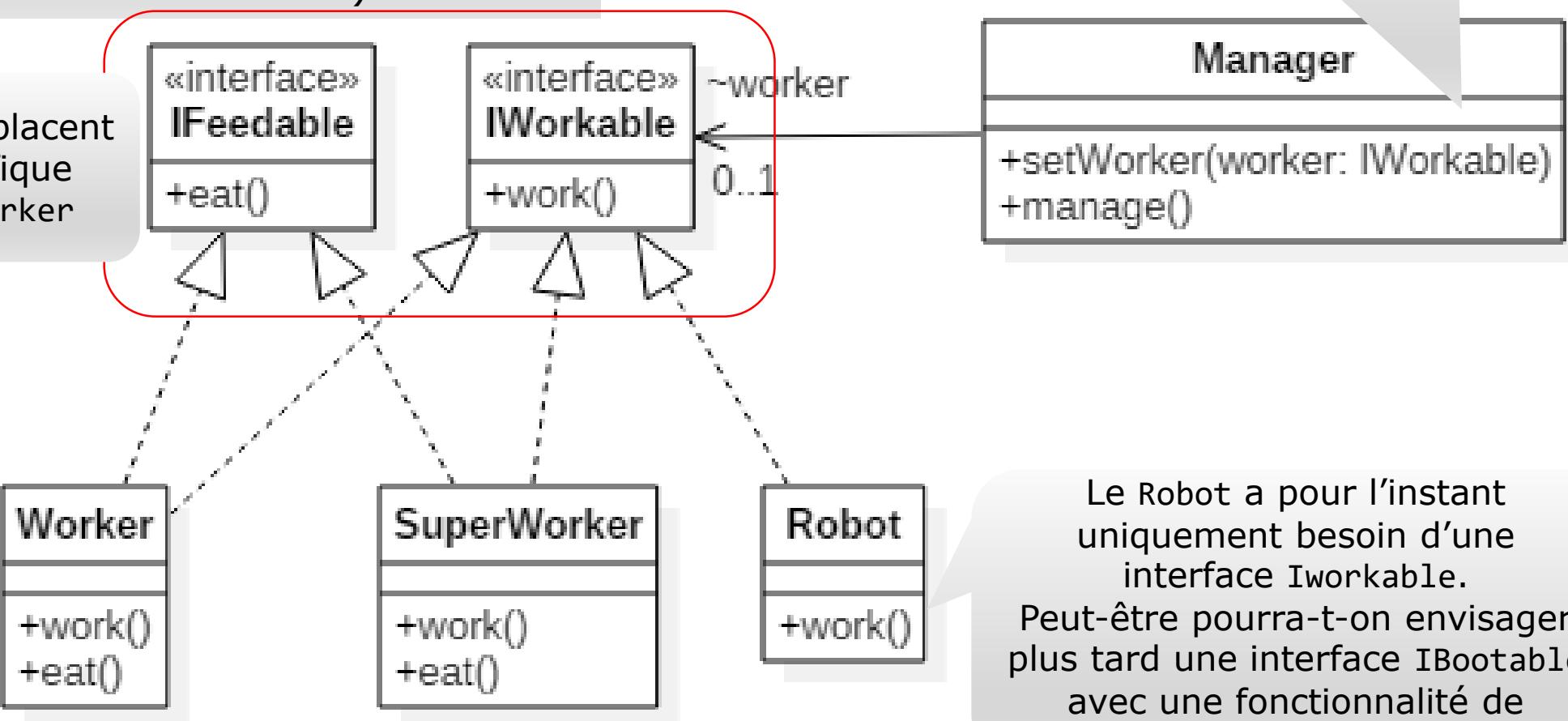
et éviter une interface trop générale

=>

Séparation de l'interface commune
en deux interfaces spécifiques !

(IWorkable et IFeedable)

Ces deux interfaces remplacent
de manière plus spécifique
l'interface massive IWorker



Seul le côté travail (Iworkable)
intéresse, pour l'instant,
le manager

Le Robot a pour l'instant
uniquement besoin d'une
interface Iworkable.
Peut-être pourra-t-on envisager
plus tard une interface IBootable
avec une fonctionnalité de
redémarrage ?

Illustration du principe ISP: Une interface commune ?

NON !!! Séparons (4/4)

Ce qui implique comme changement au niveau du code

```
public interface IWorkable {  
    public void work();  
}
```

```
public interface IFeedable {  
    public void eat();  
}
```

```
public class Worker implements IWorkable, IFeedable {  
    public void work() {  
        // ....working  
    }  
  
    public void eat() {  
        // ..... eating in launch break  
    }  
}
```

```
public class RobotWorker implements IWorkable {  
    public void work() {  
        // ... working  
    }  
}
```

```
public class Manager {  
    IWorkable worker; ←  
  
    public void setWorker(IWorkable w) {  
        worker=w;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}
```

```
public class SuperWorker implements IWorkable, IFeedable{  
    public void work() {  
        //.... working much more  
    }  
  
    public void eat() {  
        //.... eating in launch break  
    }  
}
```

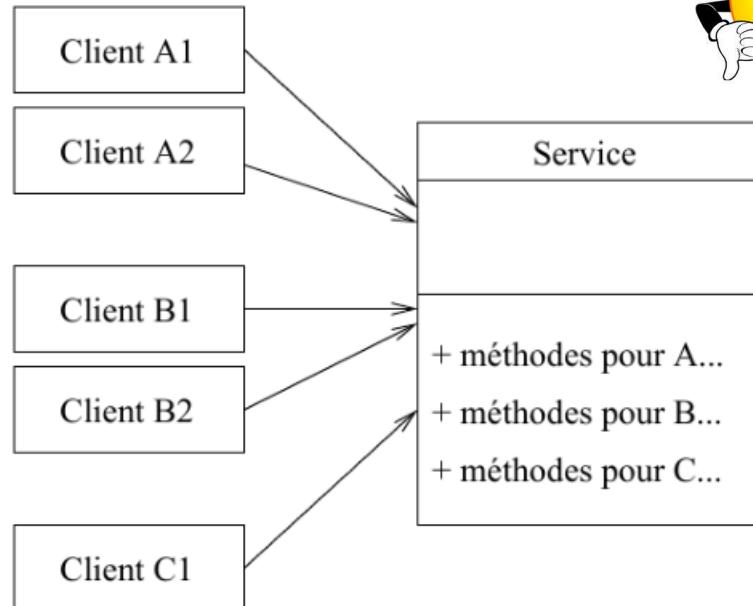
```
public interface IWorker extends IWorkable, IFeedable {  
    // possibilité de déclarer cette interface à partir des  
    // interfaces spécifiques, mais non utilisée ici, donc a  
    // priori inutile pour l'instant ;-) ...  
}
```

Principe ISP : Du problème à la solution...

Problème :

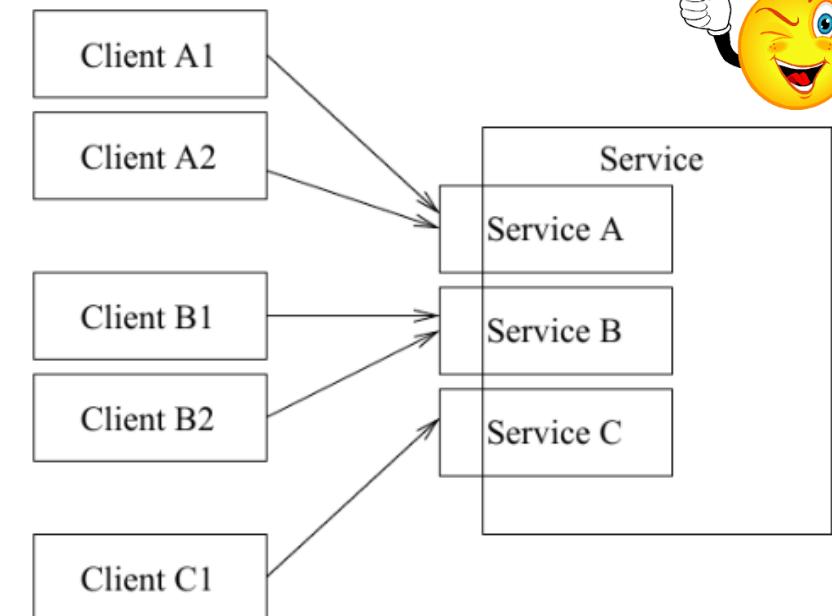
Pollution d'interface par agrégation de services :

classes qui rendent plusieurs services simultanément



Solution : La séparation !

Séparer les interfaces pour les adapter aux besoins du client
N'offrir aux classes clientes qu'un accès limité

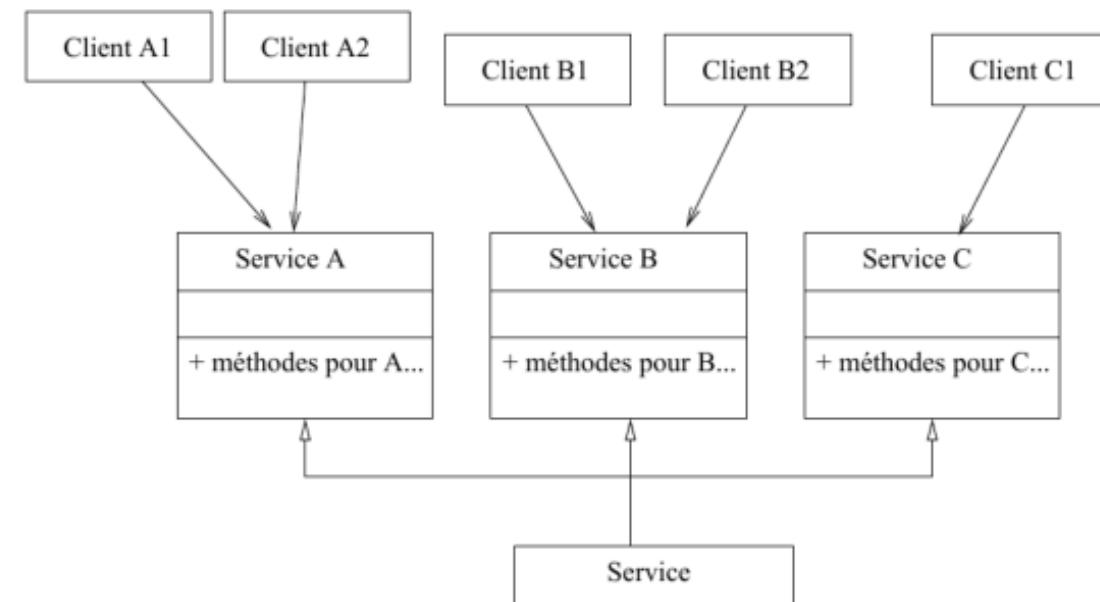


Principe de ségrégation (séparation) des interfaces : Pourquoi ?

- pour éviter qu'un client voit un service qui ne le concerne pas
- pour éviter que les évolutions dues à une partie du service aient un impact sur un client alors qu'il n'est pas concerné

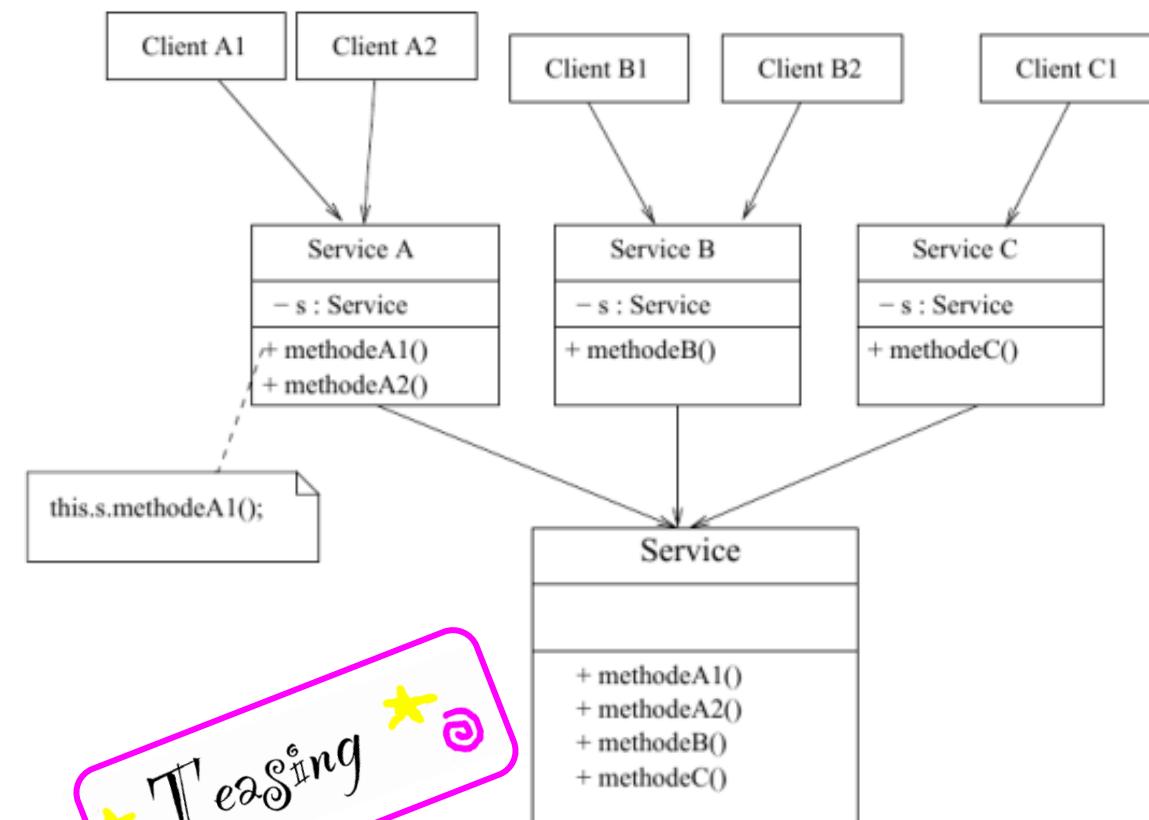
Principe ISP : Exemples de possible mise en œuvre ...

Séparation par héritage multiple (en java à l'aide d'interfaces)



Découper en parties les plus petites possibles et composables entre elles. Et laissez à votre client le soin de les assembler comme bon lui semble...

Séparation via le pattern Adapter (services représentés par des classes d'adaptation)



Teasing ☺ *

ISP: Quelques remarques ...

« Le but du **principe ISP** est **d'utiliser les interfaces pour définir des contrats**, des ensembles de fonctionnalités répondant à un besoin fonctionnel, plutôt que de se contenter d'apporter de l'abstraction à nos classes. . Il en découle une **réduction du couplage**, les **clients dépendant uniquement des services qu'ils utilisent** »



Dérive ISP

« Un des travers de ce principe peut être de multiplier les interfaces. En poussant cette idée à l'extrême, nous pouvons imaginer une interface avec une méthode par client. Bien entendu, l'expérience, le pragmatisme et le bon sens sont nos meilleurs alliés dans ce domaine. »

SRP vs ISP

- **SRP**, porte plus sur la stratégie d'**encapsulation** du détail d'**implémentation**
- **ISP**, porte plus sur la stratégie de **regroupement et d'isolation** des **contrats d'interface**, c'est à dire des méthodes publiques.

ISP en pratique et en vidéo !



Publié par Sarah Buisson
Il y a 6 mois · 1 minute · Back, Craft

SCREENCAST SUR INTERFACE SEGREGATION PRINCIPLE

Dans cette vidéo, je vais illustrer la mise en place de l'un des principes **SOLID** : The Interface Segregation Principle.



Extrait : <http://blog.xebia.fr/2017/01/18/screencast-sur-interface-segregation-principle/>

Kata de refactoring ISP
À visualiser sur : <https://www.youtube.com/watch?v=4D3AkCg6x7I>
(5 minutes 27)

S.O.L.I.D

DIP



DEPENDENCY INVERSION PRINCIPLE
Est-ce que vous souderiez directement
un branchement électrique dans un mur ?

Principe d’Inversion des Dépendances

Image : <http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

DIP décrit par Robert C Martin <https://drive.google.com/file/d/0BwhCYaYDn8EgMjdIMWlzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRINTQ3ZGMz/view>
accessible depuis <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Isabelle BLASQUEZ

Les principes SOLID : DIP (Principe d'Inversion des Dépendances)



DEPENDENCY INVERSION PRINCIPLE
Est-ce que vous souderiez directement un branchement électrique dans un mur ?

1. Les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau.
Les deux doivent dépendre d'abstractions.

2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

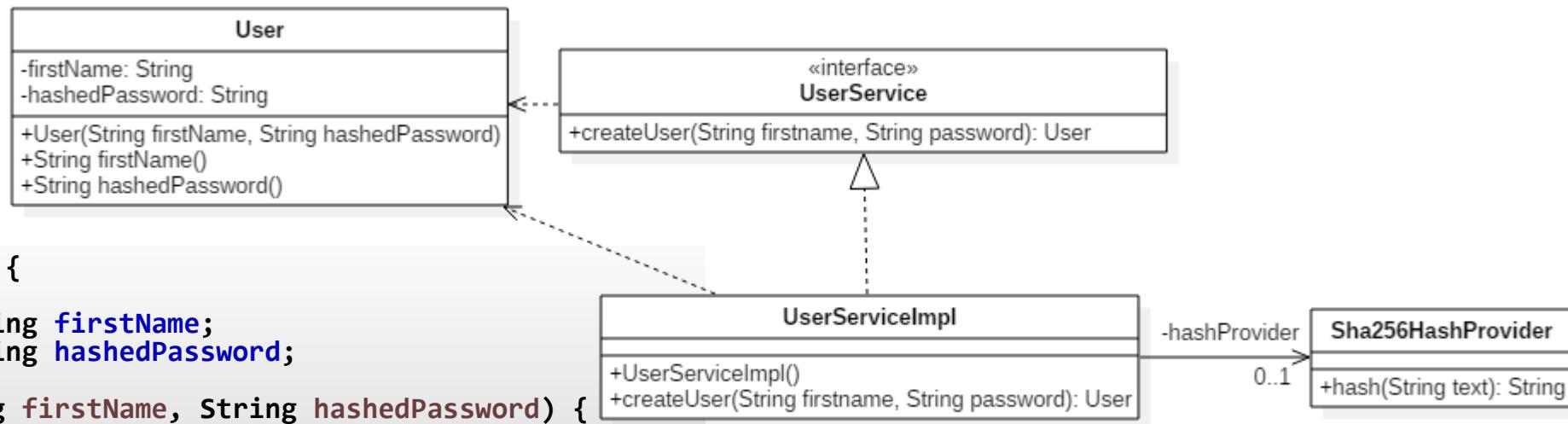
... c-a-d qu'il faut dépendre des abstractions, pas des implémentations

But : diminuer les dépendances entre objets

Moyen : placement d'interfaces intermédiaires (ou de classes abstraites)

Illustration du principe DIP: un cryptage bien découplé ? (1/3)

```
public class User {  
  
    private final String firstName;  
    private final String hashedPassword;  
  
    public User(String firstName, String hashedPassword) {  
        this.firstName = firstName;  
        this.hashedPassword = hashedPassword;  
    }  
  
    public String firstName() {return firstName;}  
  
    public String hashedPassword() {return hashedPassword;}  
}  
  
  
public class Sha256HashProvider {  
  
    public String hash(String text) {  
        try {  
            MessageDigest md = MessageDigest.getInstance("SHA-256");  
            return DatatypeConverter.printBase64Binary(md.digest(text.getBytes("UTF-8")));  
        } catch (IOException | NoSuchAlgorithmException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```



```
public interface UserService {  
    public User createUser(String firstName, String password);  
}
```

```
public class UserServiceImpl implements UserService {  
  
    private final Sha256HashProvider hashProvider;  
  
    public UserServiceImpl() {  
        hashProvider = new Sha256HashProvider();  
    }  
  
    @Override  
    public User createUser(String firstName, String password) {  
        String hashedPassword = hashProvider.hash(password);  
        return new User(firstName, hashedPassword);  
    }  
}
```

Illustration du principe DIP: un cryptage bien découplé ? (2/3)

**Exemple d'utilisation de ce code (au travers d'un test) :
Création d'un utilisateur avec un mot de passe correctement haché**

```
public class UserServiceImplTest {  
  
    @Test  
    public void should_create_user_with_hashed_password() {  
        // given  
        UserService userService = new UserServiceImpl();  
  
        // when  
        User user = userService.createUser("Bob", "secret");  
  
        // then  
        assertThat(user.firstName(), is("Bob"));  
        assertThat(user.hashedPassword(), is("K7gNU3sdo+OL0wNhqoVWhr3g6s1xYv72oL/pe/UnoLs="));  
    }  
}
```

Illustration du principe DIP: un cryptage bien découplé ?

La hachage du mot de passe se fait par un Provider qui permet d'effectuer le hash (3/3)

Pourquoi y-a-t-il Violation du principe ISP ?

```
public class UserServiceImpl implements UserService {  
    private final Sha256HashProvider hashProvider;  
  
    public UserServiceImpl() {  
        hashProvider = new Sha256HashProvider();  
    }  
  
    @Override  
    public User createUser(String firstName, String password) {  
        String hashedPassword = hashProvider.hash(password);  
        return new User(firstName, hashedPassword);  
    }  
}
```

private final HashProvider hashProvider

**public UserServiceImpl(HashProvider hashProvider) {
 this.hashProvider = hashProvider;
}**

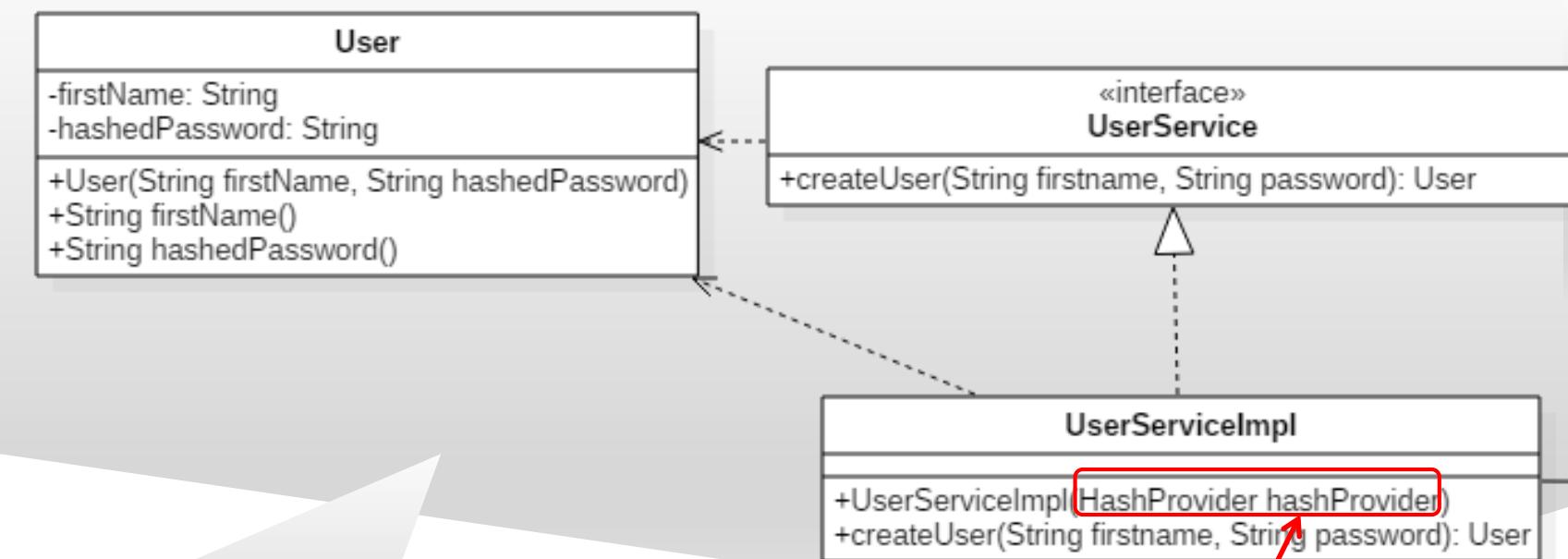
Injection de dépendance

Le provider de hash (considéré comme bas niveau car relatif aux *détails techniques*)
ne respecte pas DIP

car il dépend directement d'une implémentation et non d'une abstraction !

Pour respecter DIP, comme « *Les détails doivent dépendre des abstractions* »,
on souhaiterait plutôt voir dépendre d'une **interface HashProvider**
et que **public class Sha256HashProvider implements HashProvider**

Illustration du principe DIP: un cryptage bien découplé !!!



« Haut niveau » (plutôt fonctionnel)

2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Injection de l'abstraction
Pour ne pas instancier
directement la dépendance !

« Bas niveau » (plutôt technique) :
ajout d'une abstraction sous forme d'interface

1. Les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau.
Les deux doivent dépendre d'abstractions.

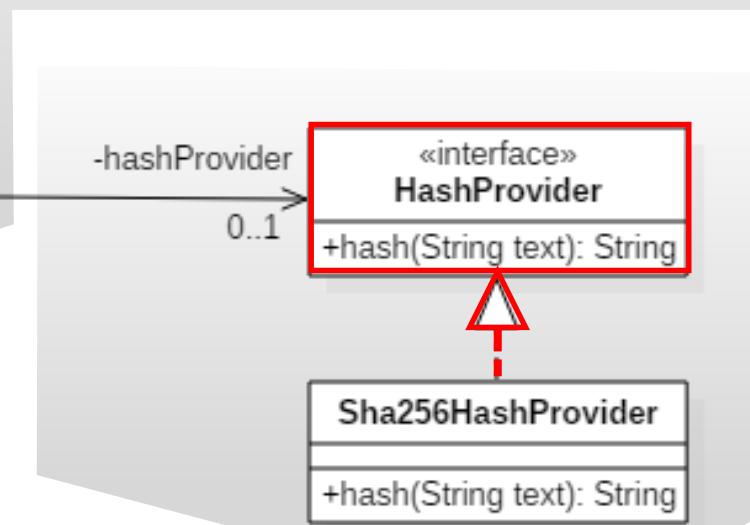


Illustration du principe DIP: un cryptage bien découplé !!!

Pour tester la création d'un utilisateur avec un mot de passe correctement haché avec un provider abstrait, il faut maintenant utiliser un mock pour simuler le comportement de l'implémentation ...

```
public class UserServiceImplTest {  
  
    @Test  
    public void should_create_user_with_hashed_password() {  
  
        HashProvider hashProvider = mock(HashProvider.class);  
        when(hashProvider.hash("secret")).thenReturn("hash");  
        UserService userService = new UserServiceImpl(hashProvider);  
  
        User user = userService.createUser("Bob", "secret");  
  
        assertThat(user.firstName(), is("Bob"));  
        assertThat(user.hashedPassword(), is("hash"));  
    }  
}
```

Injection de l'abstraction via un mock

... et écrire une méthode de test pour vérifier le bon cryptage Sha256...

```
public class Sha256HashProviderTest {  
  
    @Test  
    public void should_hash_text_using_sha256_and_output_as_base64() {  
        Sha256HashProvider hashProvider = new Sha256HashProvider();  
        String hash = hashProvider.hash("text");  
        assertThat(hash, is("mC2ePrml9VnmM/TRLN7zdh2Qn1o7ZH0ahR/q1nwyydE="));  
    }  
}
```

Illustration du principe DIP : autre exemple ...

```
class PasswordReminder {  
    private $dbConnection;  
  
    public function __construct(MySQLConnection $dbConnection) {  
        $this->dbConnection = $dbConnection;  
    }  
}
```

Violation du principe DIP

... et du principe OCP !

Vers un code plus **SOLID** ...

```
interface DBConnectionInterface {  
    public function connect();  
}
```

```
class MySQLConnection implements DBConnectionInterface {  
    public function connect() {  
        return "Database connection";  
    }  
}  
  
class PasswordReminder {  
    private $dbConnection;  
  
    public function __construct(DBConnectionInterface $dbConnection) {  
        $this->dbConnection = $dbConnection;  
    }  
}
```

DIP en pratique et en vidéo !

SCREENCAST : DEPENDENCY INVERSION PRINCIPLE

Dans ce screencast, nous réalisons un kata pour appliquer le Principe d'Inversion de Dépendance (Dependency Inversion Principle – DIP) de SOLID. Dans la programmation orientée objet, ce principe consiste à découpler les composants de haut niveau (la logique de l'application) des composants de bas niveau (les implementations). Lorsque ce principe est appliqué, cela rend l'application moins dépendante de ses implémentations et ainsi plus simple à maintenir et à faire évoluer.

Le DIP est normalement très visible dans les applications qui appliquent l'architecture Hexagonale ou même dans l'[architecture Onion](#). C'est pourquoi nous avons passé un peu de temps à la fin de ce screencast à isoler les composants du domaine.

Extrait : <http://blog.xebia.fr/2015/02/25/screencast-dependency-inversion-principle/>

Kata de refactoring DIP

À visualiser sur : <https://www.youtube.com/watch?v=wGqQ1UCIVnI> (15 minutes)

Code source disponible sur <https://github.com/xebia-france/geocoder-kata>
→ branche **master** contient le problème (état initial)
→ branche **solution** contient les évolutions présentées dans la vidéo



SOLID

Software Development is not a Jenga game



SINGLE RESPONSIBILITY PRINCIPLE

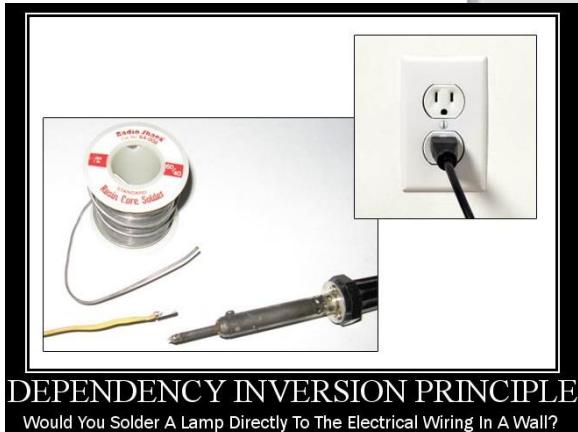
Just Because You Can, Doesn't Mean You Should



OPEN CLOSED PRINCIPLE

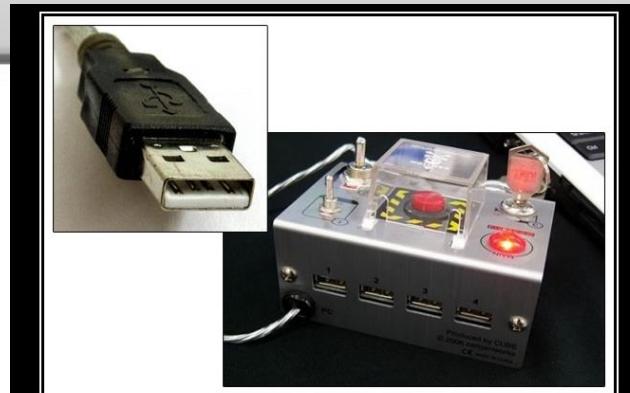
Open Chest Surgery Is Not Needed When Putting On A Coat

S . O . L . I . D



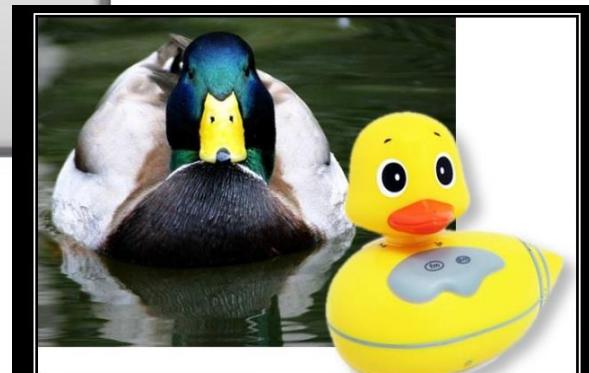
DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Concevoir, c'est faire des choix... et faire des choix demande de prendre du recul et de s'interroger

SRP

Une classe doit avoir une et une seule responsabilité

⇒ chaque classe a-t-elle une responsabilité qui lui est propre ?

OCP

***Une classe doit être ouverte aux extensions,
mais fermée aux modifications.***

⇒ En cas d'extension, le code doit-il être édité pour être modifié ?
Y-a-t-il une abstraction dans le code, pourrait-il y en avoir une ?

LSP

Les sous-types doivent pouvoir être substitués à leurs types de base

⇒ S'il y a un héritage, est-il bien conçu ?
Les sous-classes peuvent-elles remplacer leurs classe de base sans violation de contrat ?
Une délégation ne donnerait-elle pas plus de souplesse ?

ISP

Un client ne devrait jamais être forcé de dépendre d'une interface qu'il n'utilise pas

⇒ S'il y a des interfaces, ne sont-elles trop générales ? Peuvent-elles être **séparées** en interfaces plus spécifiques pour être mieux ciblées ? (*tout en ne tombant pas dans l'excès d'une interface par méthode*)

1. Les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau. Les deux doivent dépendre d'abstractions.

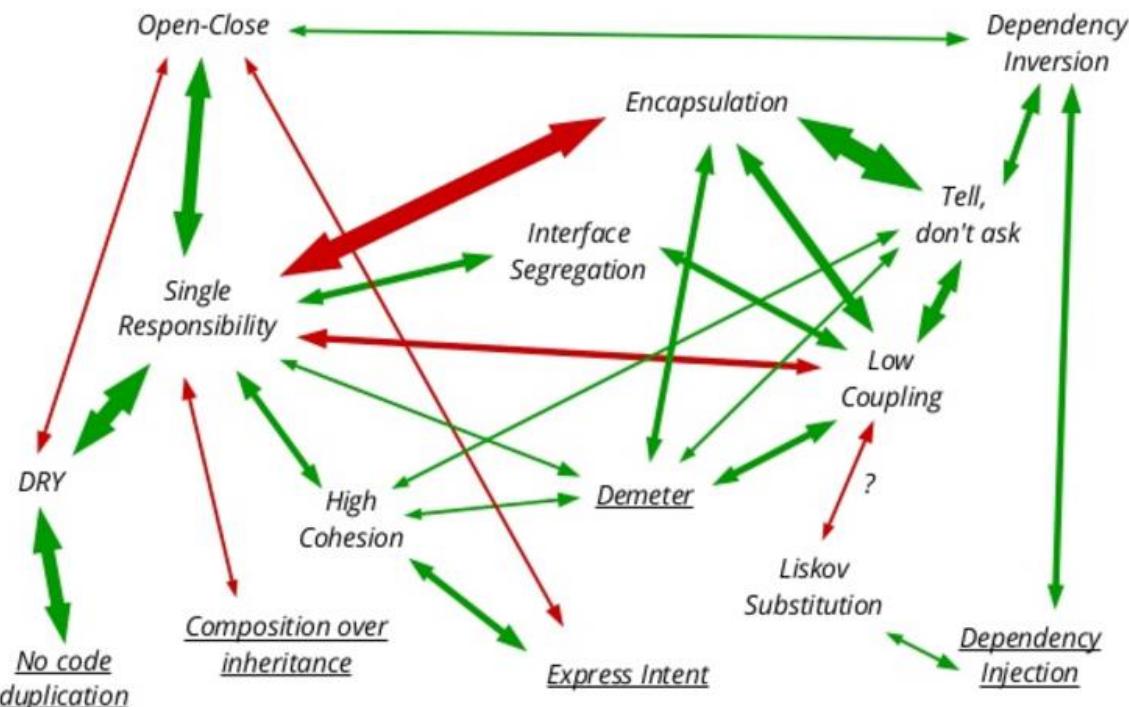
2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

⇒ Y-a-t-il assez d'abstractions (interface) pour découpler correctement les modules de haut niveau des modules de bas niveau ? Est-ce qu'on dépend des abstractions et non des implémentations ?
Le fonctionnel est-il bien indépendant du technique ?

...et faire aussi des compromis parfois !

Le pragmatisme et l'expérience vont nous permettre d'arbitrer sur une conception plus ou moins abstraite, plus ou moins tolérante au changement.

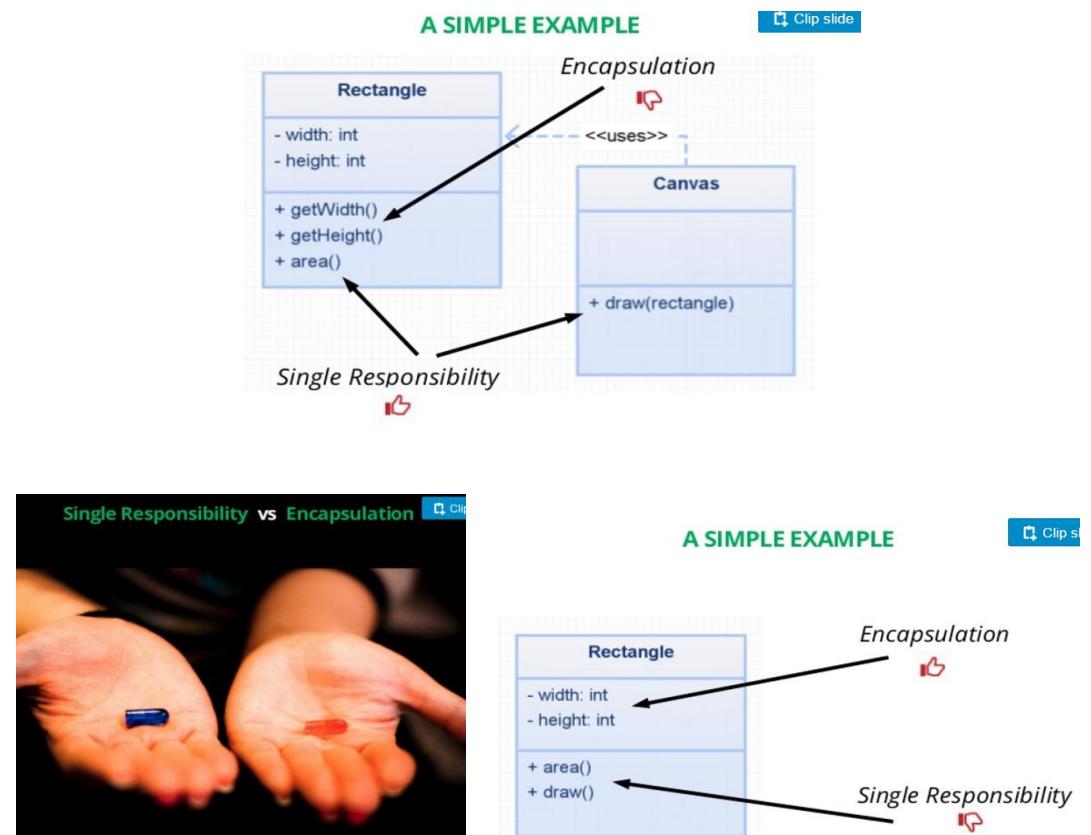
Tensions and Synergies



<https://docs.google.com/spreadsheets/d/1jiqS5sol0PCo8Rm4M6LFj20L3I6yJeyuOjLAK23RDWY/edit>

Extrait : <https://www.slideshare.net/brain79/design-principles-tensions-and-synergies-v30/16>

A voir : https://github.com/lucaminudel/tensions_and_synergies_between_design_principles



SOLID or not SOLID? ⇒ **SOLID vs YAGNI** (*Tolérance au changement et productivité*)

(à voir dans <http://blog.xebia.fr/2011/07/18/les-principes-solid/>)

Isabelle BLASQUEZ

Conclusion : A propos des principes SOLID ...

- Les principes SOLID ne sont ni des recettes miraculeuses, ni des règles strictes, mais plutôt des propositions de **bonnes pratiques**.
- Les principes SOLID placent le **contrôle des dépendances** au cœur de l'activité de **conception**
*... dans le but de limiter les coûts de modification et rendre les logiciels tolérants au changement
(avec un code plus souple, facilement maintenable, extensible et réutilisable)*
- Les principes SOLID se retrouvent dans de nombreux **Design Patterns**

Conclusion : Contribution de SOLID aux bonnes pratiques de conception

... Ou comment concevoir des logiciels maintenables et réutilisables ...

- Structuration en **package de classes**, pas de dépendances cycliques entre les packages
- **Forte cohésion**: faciliter la compréhension, gestion et réutilisation des objets en concevant des classes à but unique **(SRP)**
- **Faible couplage** : réduire l'impact des modifications en affectant les responsabilités de façon à minimiser les dépendances entre classes. **(OCP)**
- **Indirection**: *limiter le couplage* et protéger des variations en ajoutant des objets supplémentaires **(DIP)**
- **Utilisation correcte de l'héritage** : *limiter le couplage* en utilisant la composition plutôt que l'héritage pour déléguer une tâche à un objet. **(LSP)**
- **Protection des variations** : Identifier les points de variations et d'évolution et séparer ces aspects de ceux qui demeurent constants. **(OCP)**