Clean Code, SOLID

Sébastien Jean

IUT de Valence
Département Informatique

v2.6, 17 septembre 2023

IUT Valence
Université Grenoble Alpes

Du logiciel qui marche ou du logiciel qui dure ?

- Ecrire du logiciel qui marche est à la portée du premier venu ;-)
- **Mais**
  - Le logiciel doit pouvoir évoluer sur le long terme
  - Il n'y a pas de pérennité sans qualité !

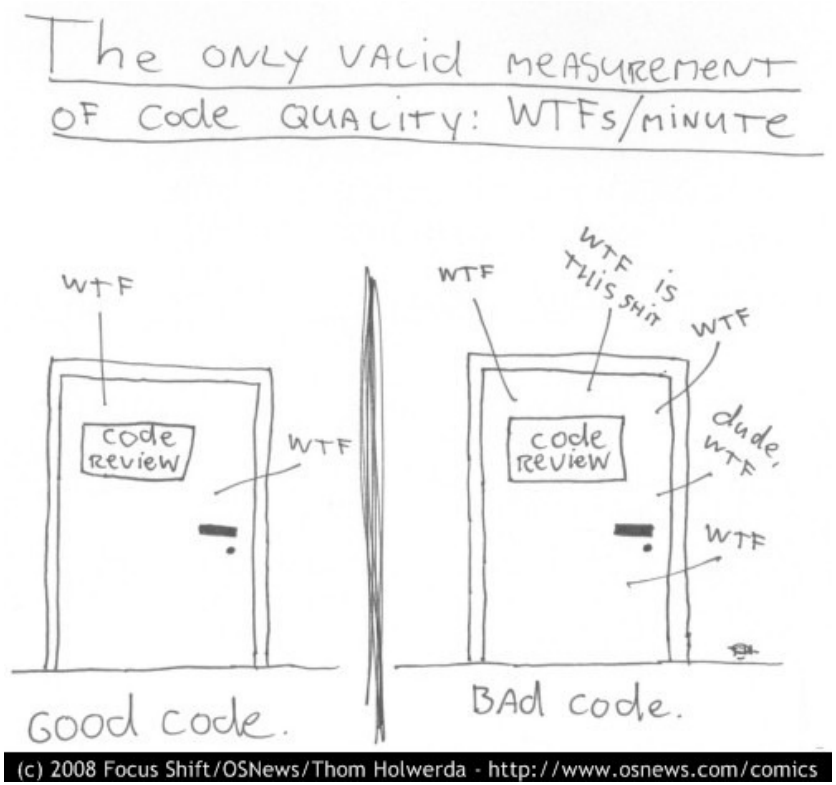
1981

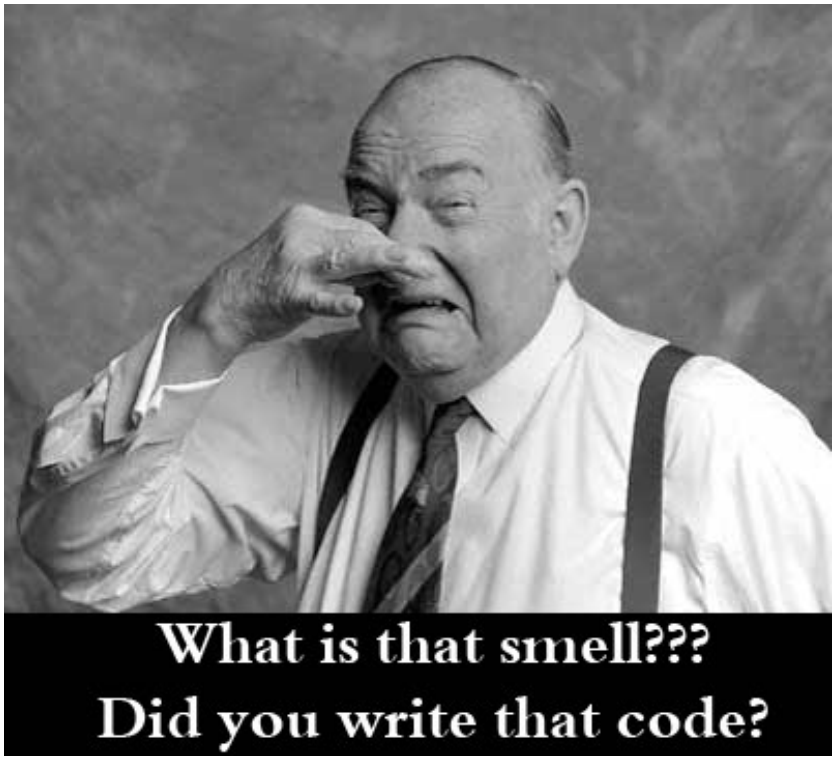
2023

IUT Valence Université Grenoble Alpes

Qu'est ce qu'un code de qualité ?

- **Conforme aux exigences** et **exempt de bug**
- **Lisible** et **facile à comprendre**, **facile à utiliser**
- **Aisément maintenable** et **réutilisable**
- **Efficace**





IUT Valence
Université Grenoble Alpes

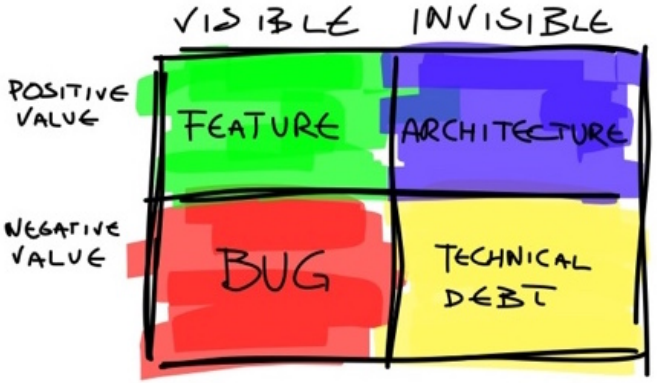
Entropie logicielle et dette technique

- **Entropie logicielle** (Lehman, 1980)

> "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

- **Dette technique** (Cunningham, 1992)

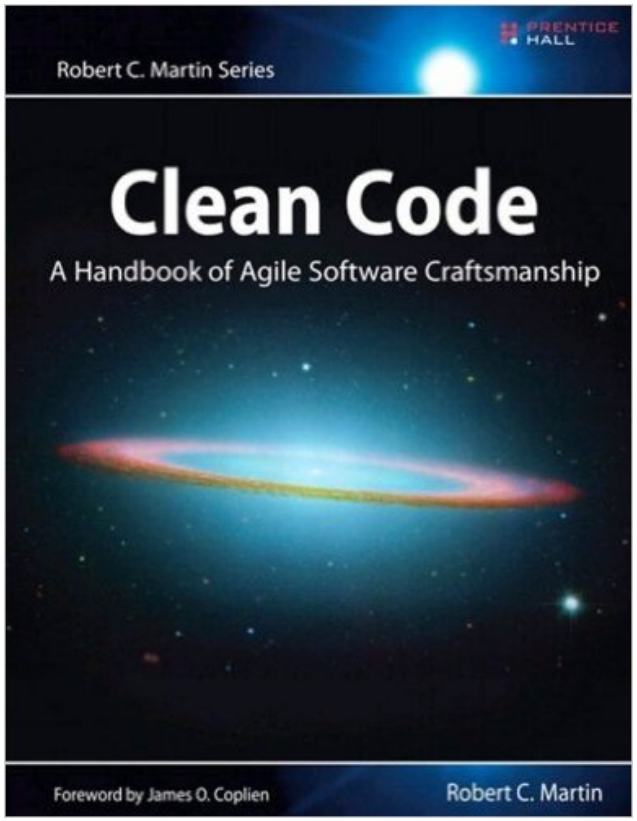
> "Shipping first time code is like going into **debt**. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as **interest** on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

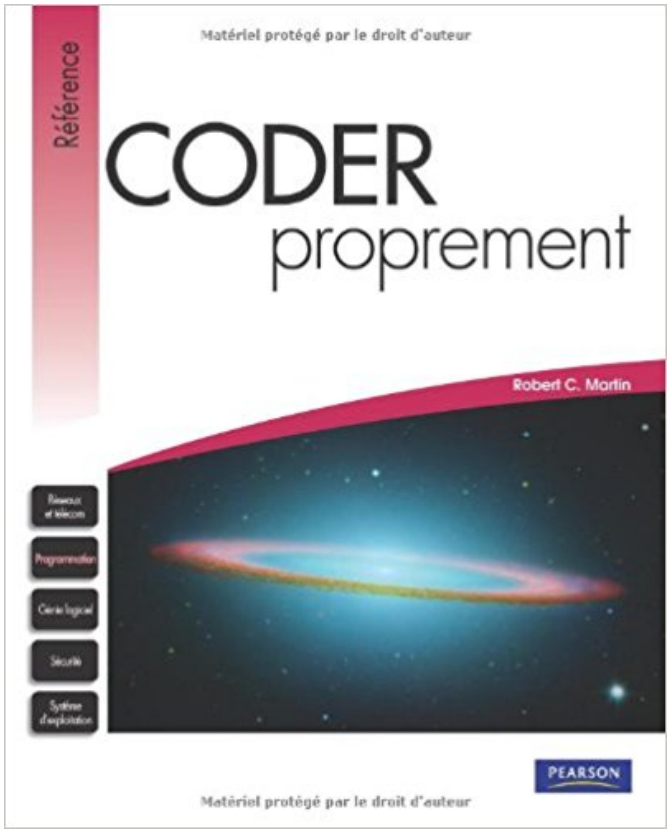


IUT Valence Université Grenoble Alpes

Clean Code

- Ouvrage de référence





IUT Valence
Université Grenoble Alpes

Clean Code

Citation (extraite de *Clean Code*)

Clean code always looks like it was written by someone who cares.

Michaël Feathers, auteur de *Working with Legacy Code*

Citation (extraite de *Clean Code*)

You know you are working with clean code when each routine you read turns out to be pretty much what you expected.

Ward Cunningham, inventeur des wikis, co-inventeur de l'*Extreme Programming*

Ce qui contribue à un code propre

- Le **nommage intentionnel**
- Le bon usage des **commentaires**
- La **mise en forme**
- des **bonnes pratiques de conception/programmation**

IUT Valence
Université Grenoble Alpes

Nommage

Code Java

```java
public List<int[]> getList ()
{
  List<int[]> l2 = new ArrayList<int[]>();

  for (int[] x : this.l1)
    if (x[0] == 4) l2.add(x);
  return l2;
}
```

IUT Valence Université Grenoble Alpes

Nommage

- Choisir des noms **révélateurs des intentions**
- Définir des constantes dont le nom véhicule l'intention

Code Java

```
public List<int[]> getDeadCells ()
{
  List<int[]> deadCells = new ArrayList<int[]>();

  for (int[] cell : this.allTheCells)
    if (cell[STATUS_OFFSET] == DEAD) deadCells.add(cell);
  return deadCells;
}
```

IUT Valence Université Grenoble Alpes

Commentaires

Code Java

```
//    Ici, un commentaire long et illisible pour expliquer
//    ce qui se passe entre 7h et 8h et entre 17h et 18h

if (((hour > 7)&&(hour < 8))||((hour > 17)&&(hour < 18)))
    ...
```

IUT Valence Université Grenoble Alpes

Commentaires

- Essayer de rendre le code le plus possible **auto-descriptif**
  - En remplaçant des expressions ou blocs d'instructions par des appels de méthodes extraites et au nom explicite

Code Java

```
if (isRushHour(hour)) ...
```

Commentaires

- **Se limiter à :**
  - Expliquer l'intention, l'implicite
  - Avertir des prérequis et conséquences

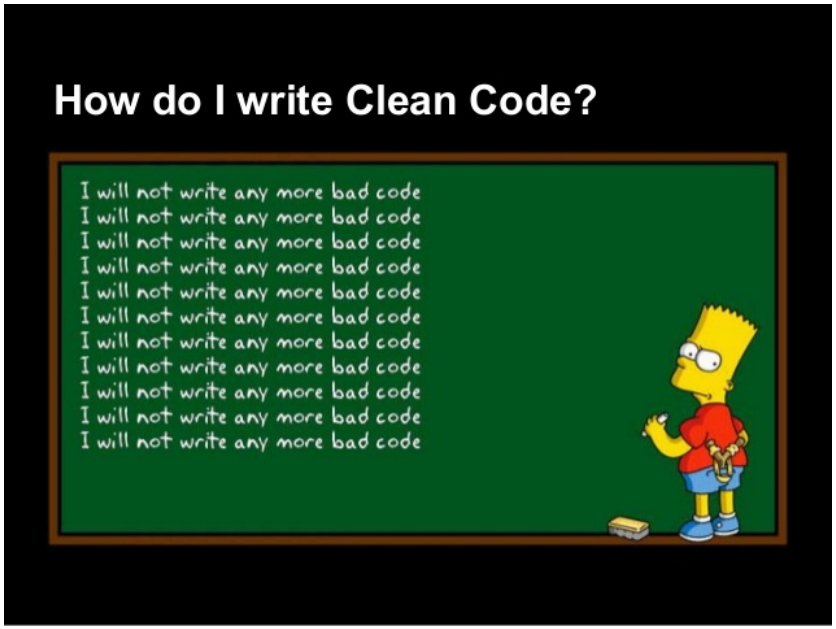
- **Ne jamais :**
  - Ecrire des commentaires redondants
    - avec le sens véhiculé explicitement les déclarations ou l'implémentation
    - avec d'autres vecteurs d'information (licences, *versioning*, ...)
  - Mettre de code en commentaire
    - c'est l'intention du *versioning*
- **Eviter de :**
  - Mettre des commentaires dans le code

IUT Valence
Université Grenoble Alpes

Mise en forme

- Les fichiers doivent être **courts** (moy. 200 lignes, max. à 500)
  - Les lignes doivent être de **taille raisonnable** (< 120 caractères)
    - Limite historique de *Hollerith* = 80 caractères
    - Autour de 150 caractères parait raisonnable aujourd'hui

- Le code doit :
  - être proprement formaté (indentation, . . .)
  - être correctement espacé
  - se lire de haut en bas
    - ***Règle de la décroissance***

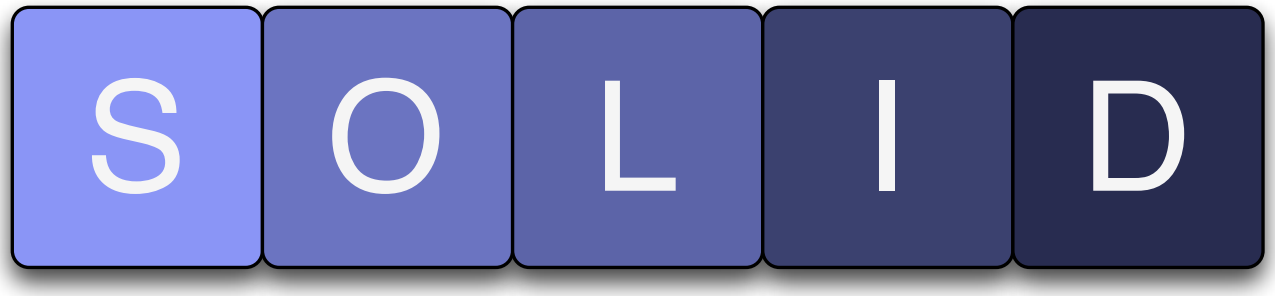


Techniques / outils pour la qualité du code

- *Odeurs de code* : métriques permettant d'alerter sur la mauvaise qualité du code
  - méthode/classe volumineuse, méthode avec trop de paramètres, ...
  - cf. outils d'analyse (*SonarLint*, ...)
- **Refactoring** : actions de transformation du code
  - extraction de variables/méthodes/sous-classe, déplacement de méthode, ...
  - cf. assistants des IDE
- cf. `https://refactoring.guru/refactoring`

IUT Valence Université Grenoble Alpes

SOLID



- 5 principes de base de COO/POO
  - **S**ingle Responsability Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

IUT Valence Université Grenoble Alpes

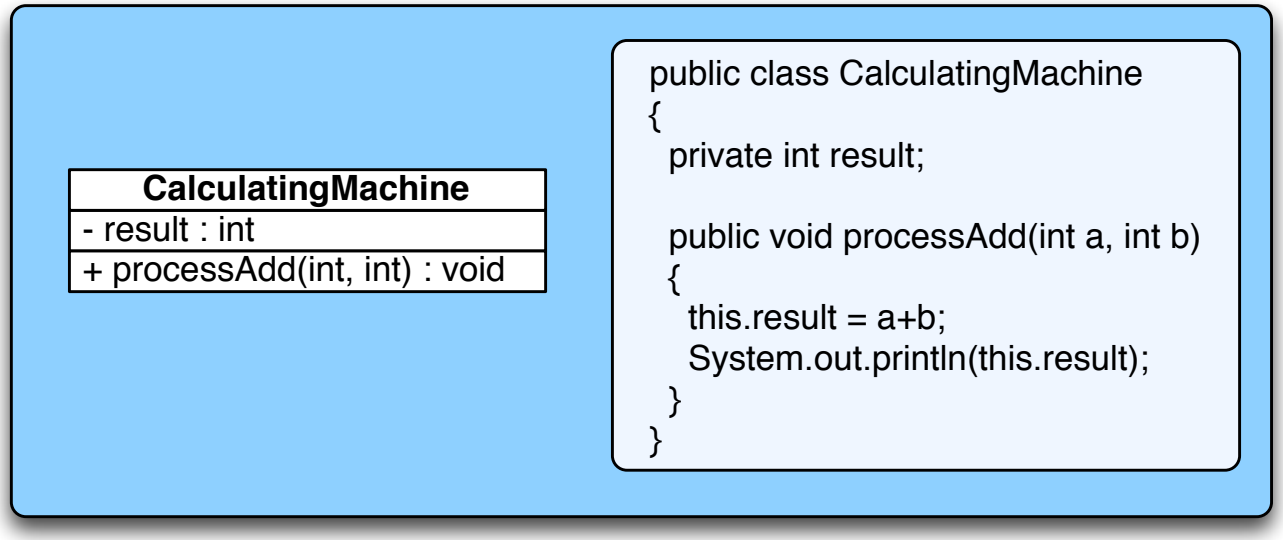
Single Responsibility Principle

- Une classe (ou une méthode) ne doit avoir **qu'une et une seule raison de changer**

IUT Valence
Université Grenoble Alpes

Une mauvaise conception ?

- Combien de responsabilités possède la classe ou sa méthode ?



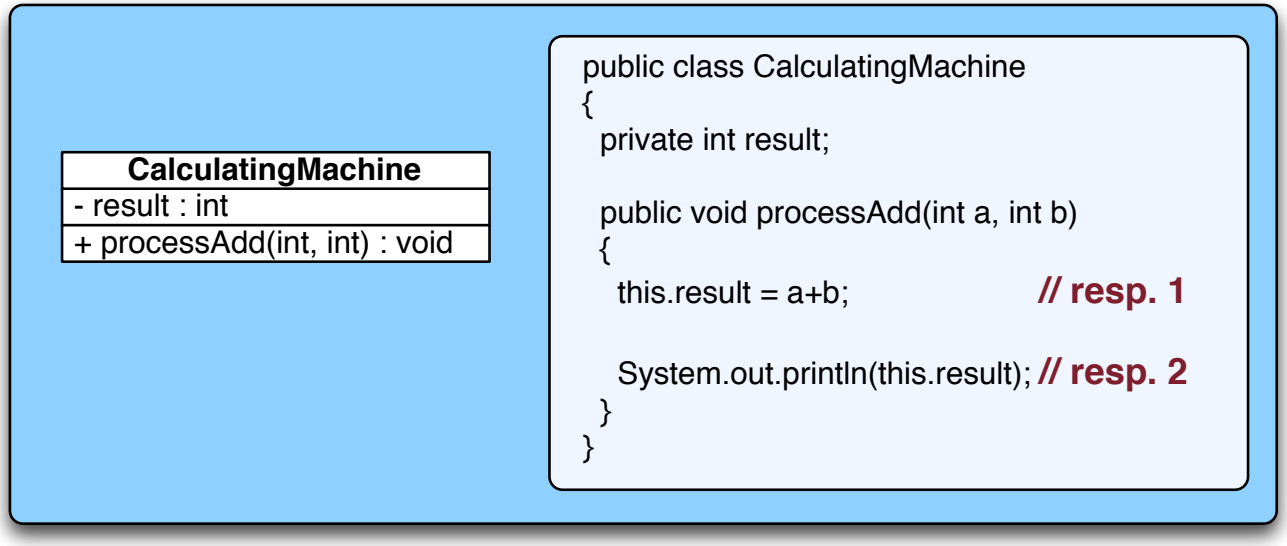
- *Exemple inspiré de*

http://www.codeproject.com/Articles/426773/Single-responsibility-principle-SRP

IUT Valence Université Grenoble Alpes

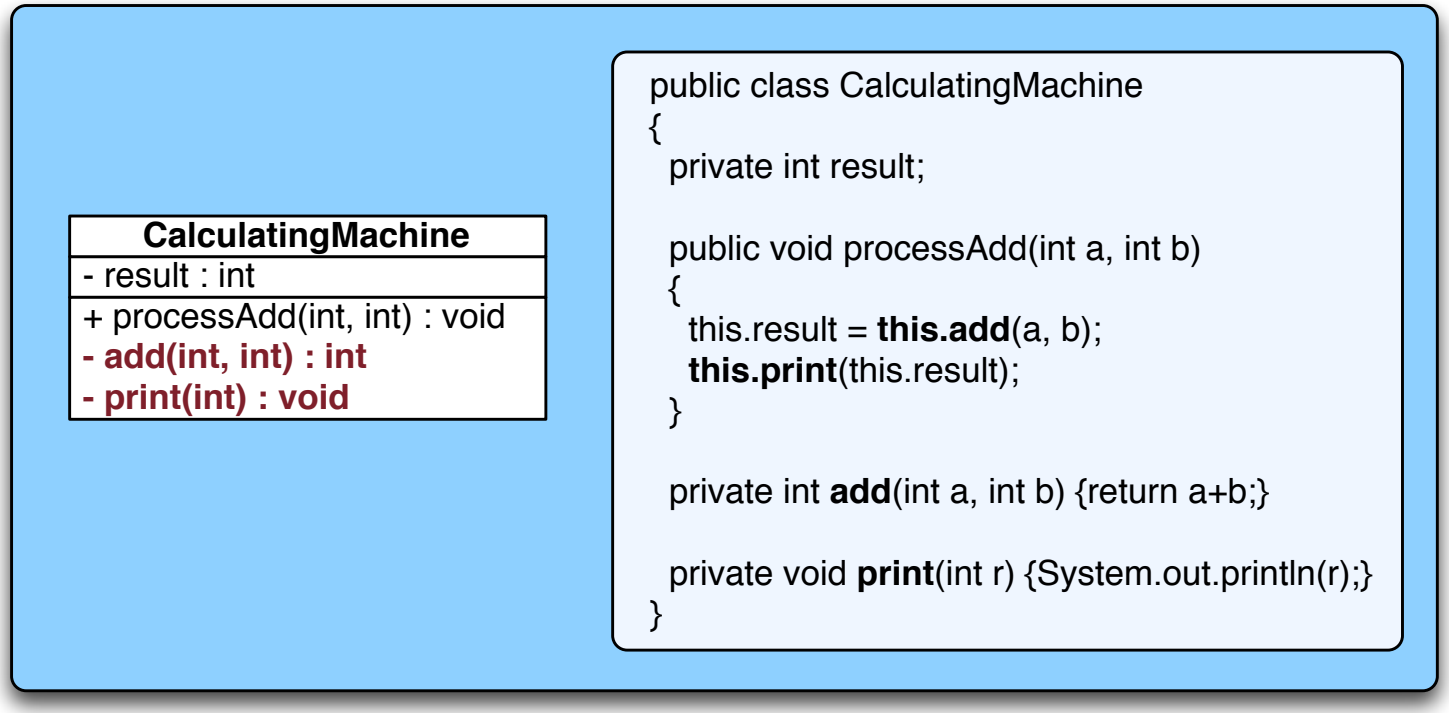
Une mauvaise conception

- La **méthode et la classe** ont **2 responsabilités** :
  - Calculer la somme
  - Afficher le résultat



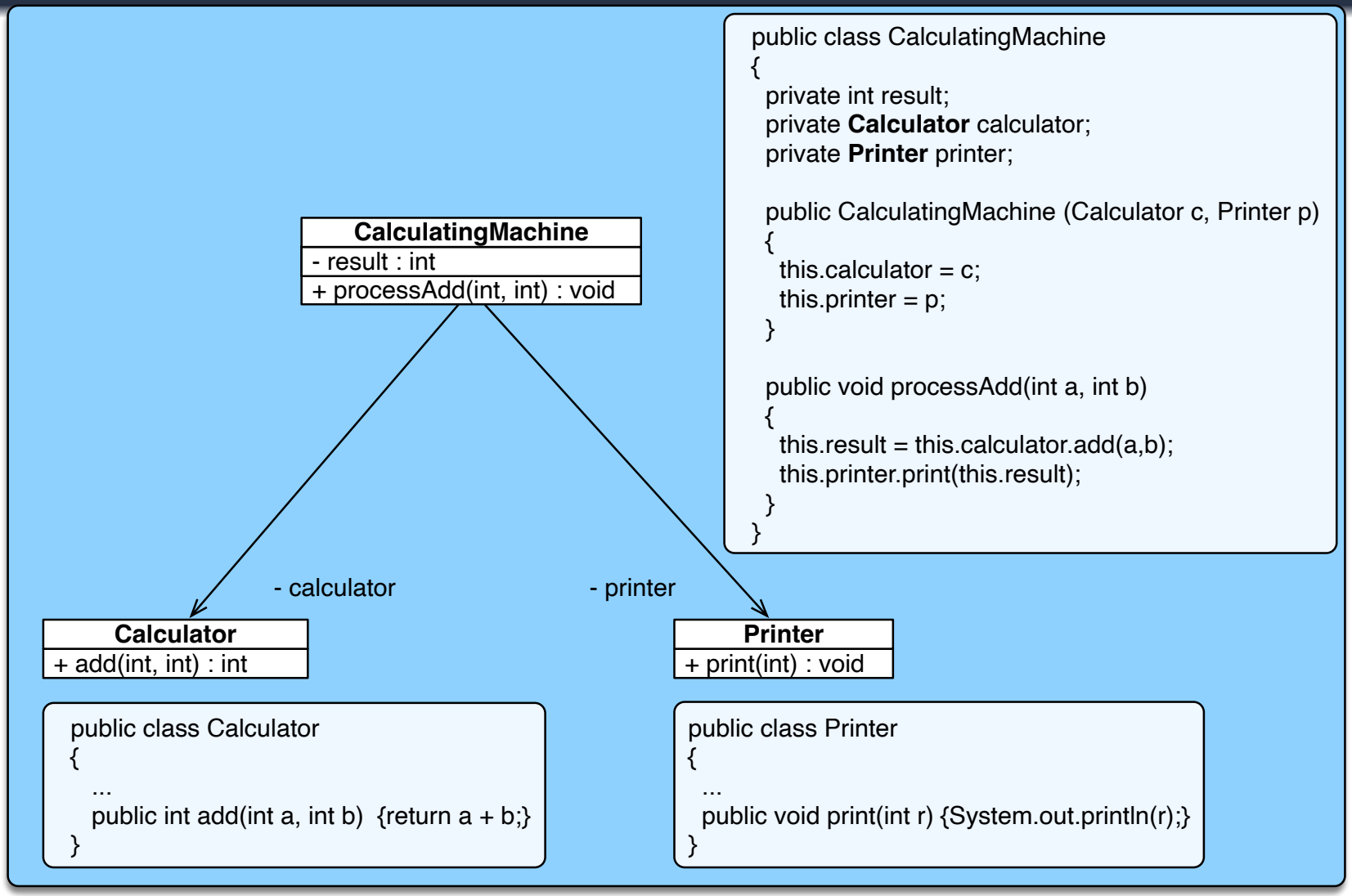
SRP en action

- La **méthode n'a plus qu'une responsabilité**
  - Coordonner le calcul du résultat et son affichage
    - pas de connaissance de l'implémentation du calcul ni de l'affichage
    - simple connaissance de l'appel et de sa sémantique



- Mais la classe a toujours les deux responsabilités !

Une meilleure conception



- La classe et la méthode n'ont plus qu'une responsabilité
- Et on pourrait encore aller plus loin ...

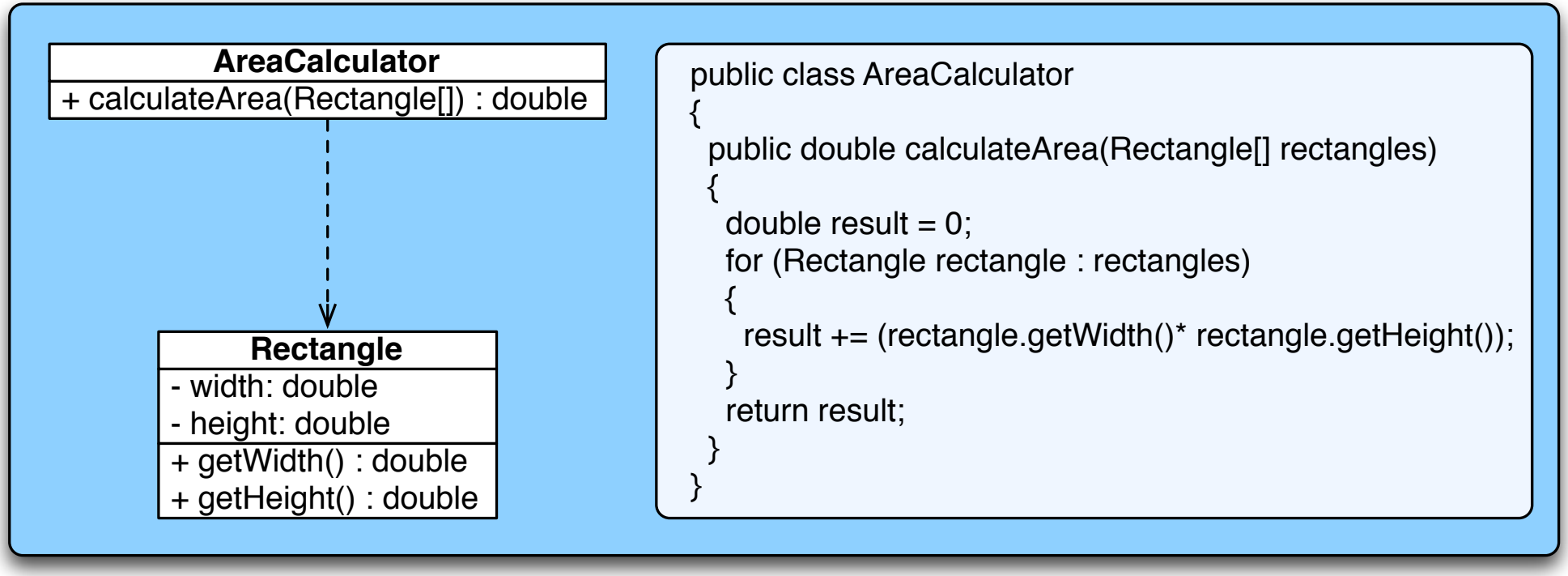
IUT Valence Université Grenoble Alpes

Open/Closed Principle

- Une classe doit être **ouverte à l'extension** mais **fermée à la modification**

IUT Valence
Université Grenoble Alpes

Exemple



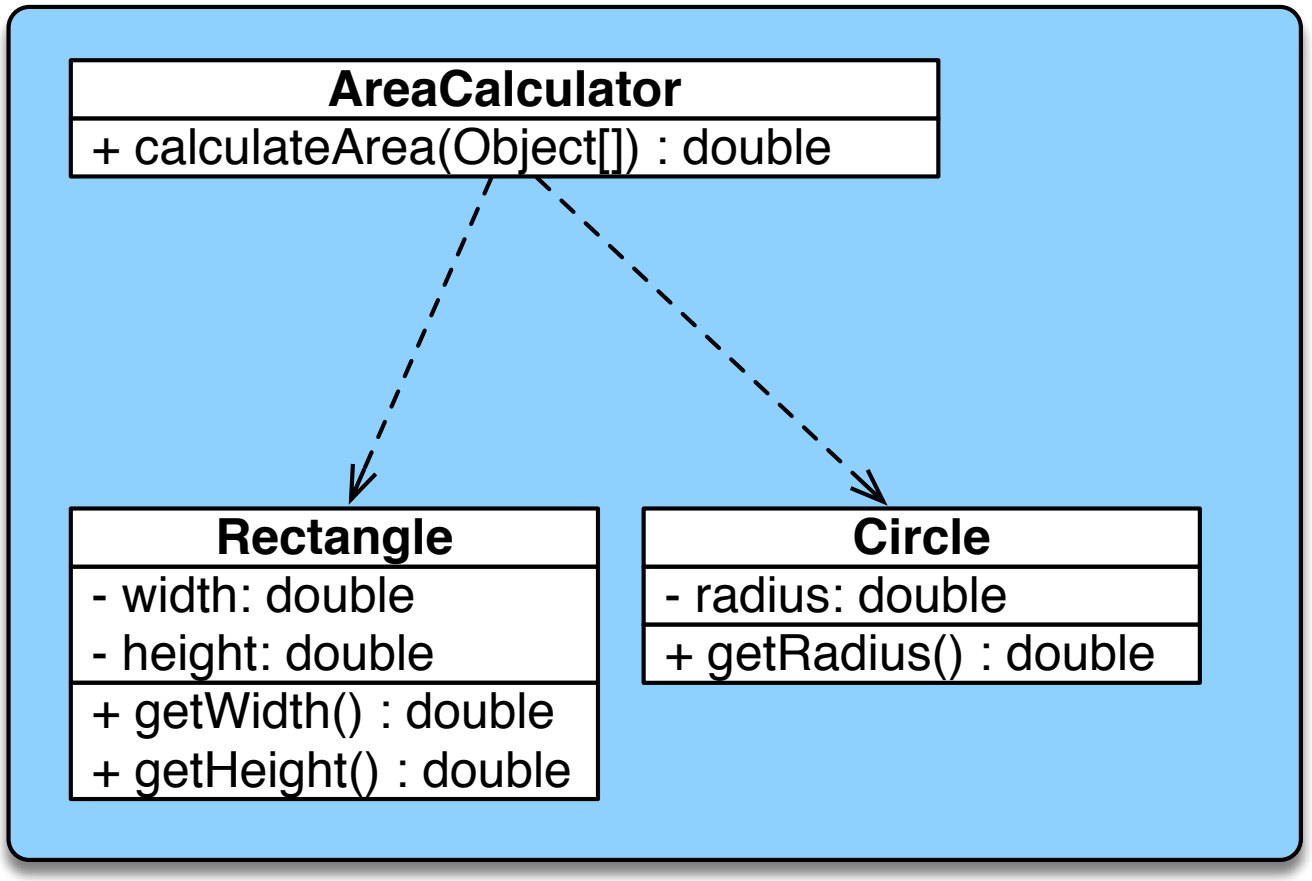

```
public class AreaCalculator
{
  public double calculateArea(Rectangle[] rectangles)
  {
    double result = 0;
    for (Rectangle rectangle : rectangles)
    {
      result += (rectangle.getWidth()* rectangle.getHeight());
    }
    return result;
  }
}
```

- *Exemple inspiré de*

  http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/

IUT Valence Université Grenoble Alpes

Prise en compte de formes différentes : une solution simple

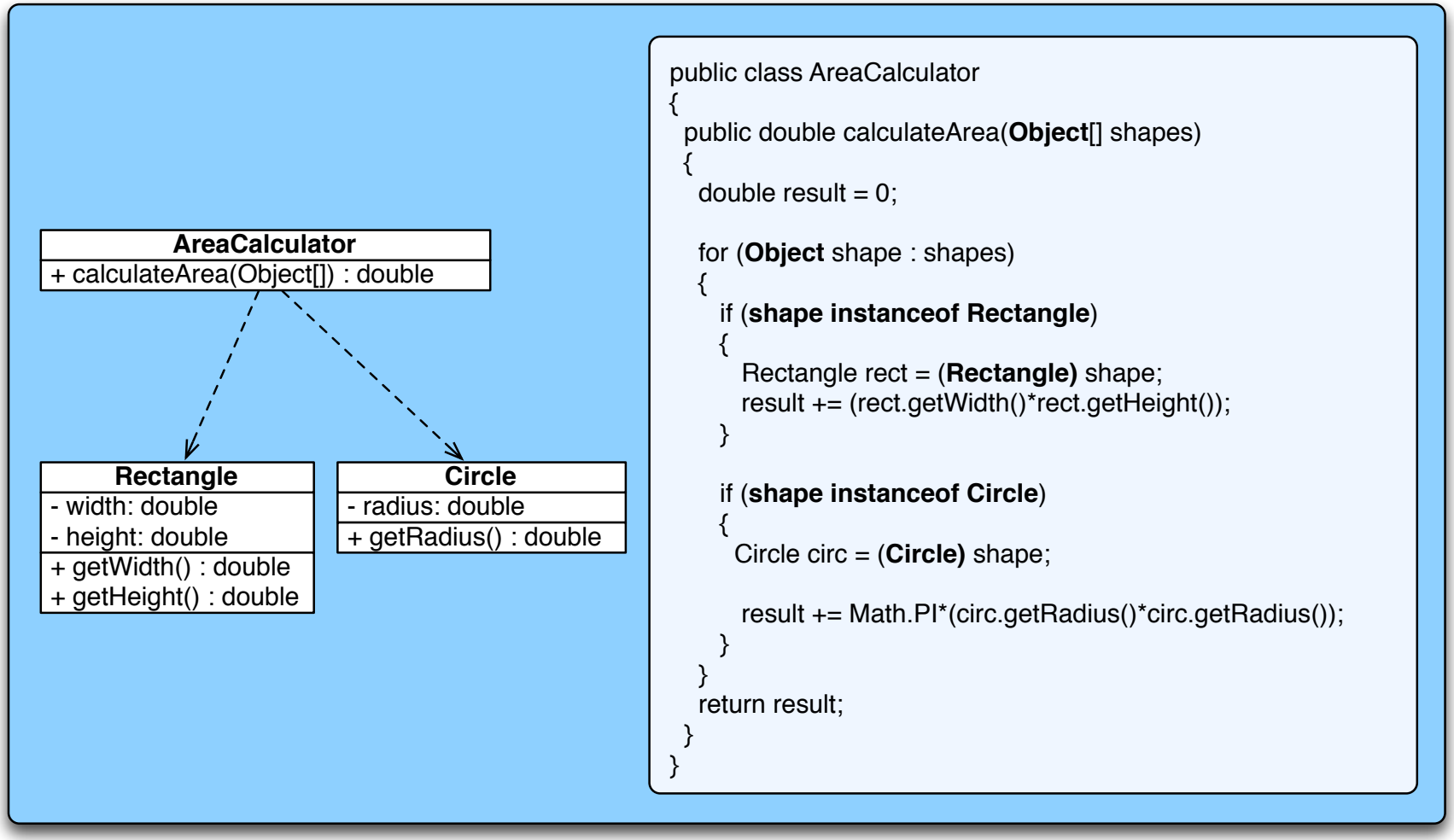


- Pourquoi est-ce une mauvaise idée ?

IUT Valence Université Grenoble Alpes

Prise en compte de formes différentes : une solution simple

- La généricité apportée par l'utilisation d'`Object` nécessite de **modifier le code de la classe** pour la prise en compte de nouvelles formes

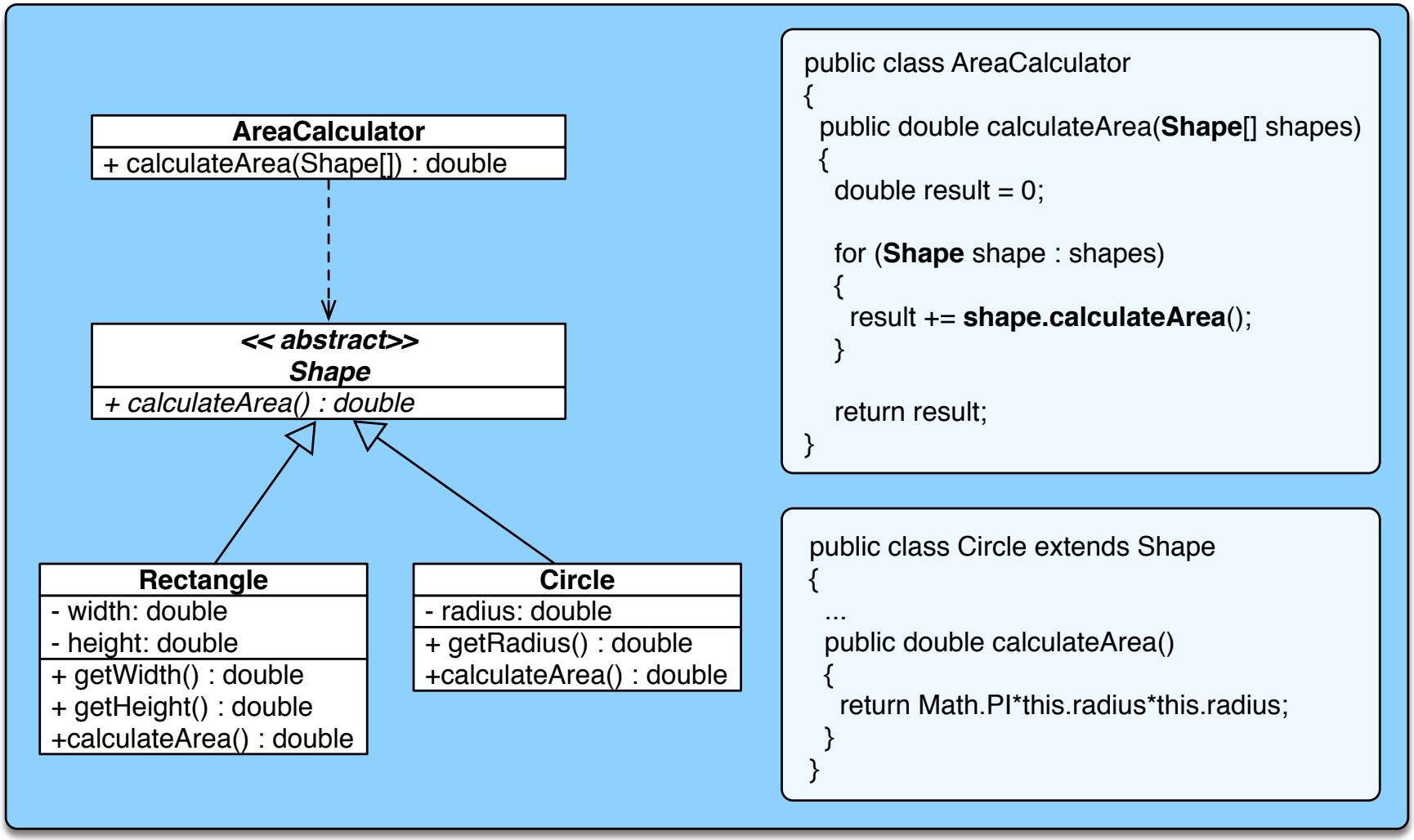



```
public class AreaCalculator
{
  public double calculateArea(Object[] shapes)
  {
    double result = 0;

    for (Object shape : shapes)
    {
      if (shape instanceof Rectangle)
      {
        Rectangle rect = (Rectangle) shape;
        result += (rect.getWidth()*rect.getHeight());
      }

      if (shape instanceof Circle)
      {
        Circle circ = (Circle) shape;

        result += Math.PI*(circ.getRadius()*circ.getRadius());
      }
    }
    return result;
  }
}
```

IUT Valence Université Grenoble Alpes

Une meilleure conception

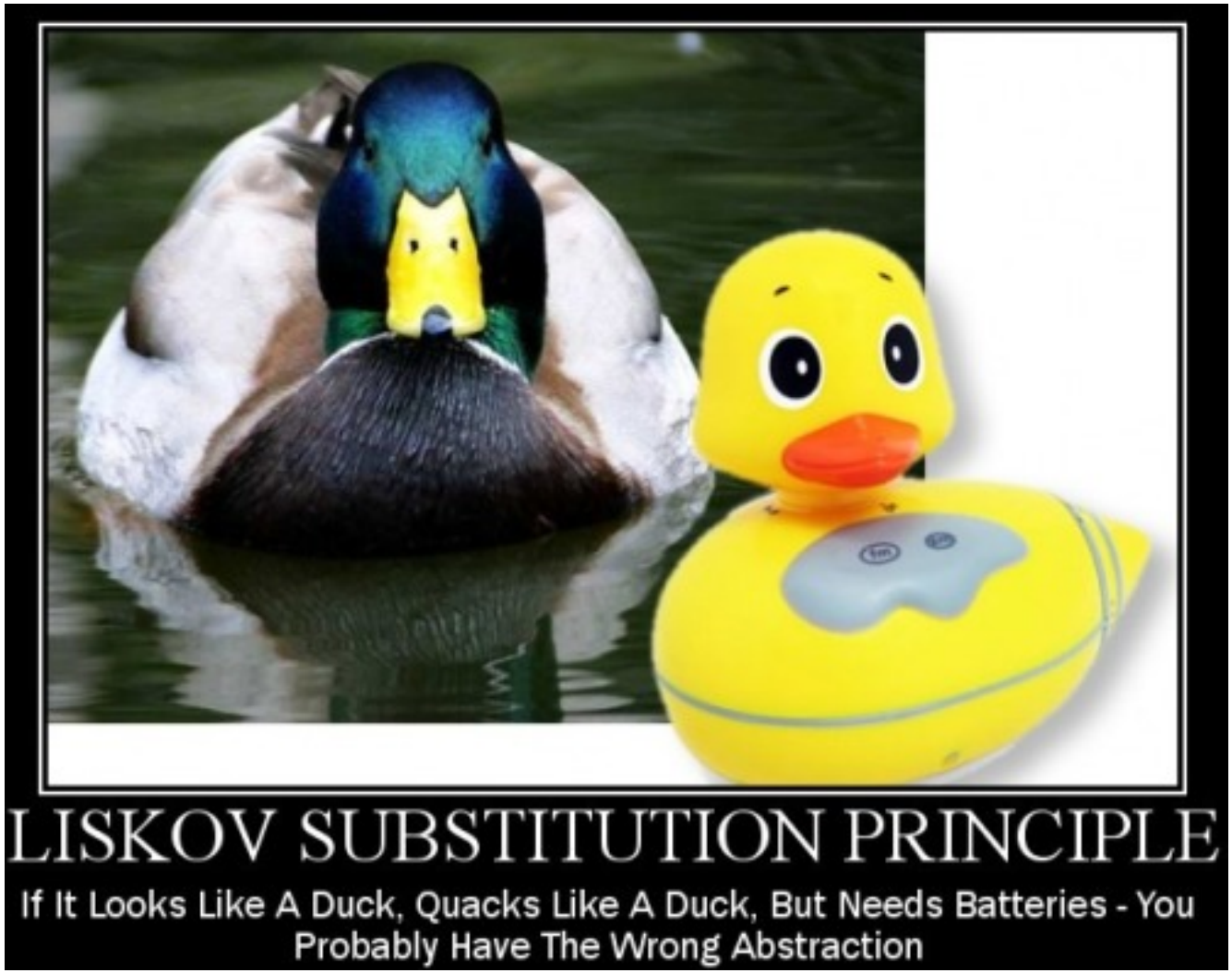
- La classe `AreaCalculator` est ouverte par l'extension (de `Shape`)



IUT Valence Université Grenoble Alpes

Liskov Substitution Principle

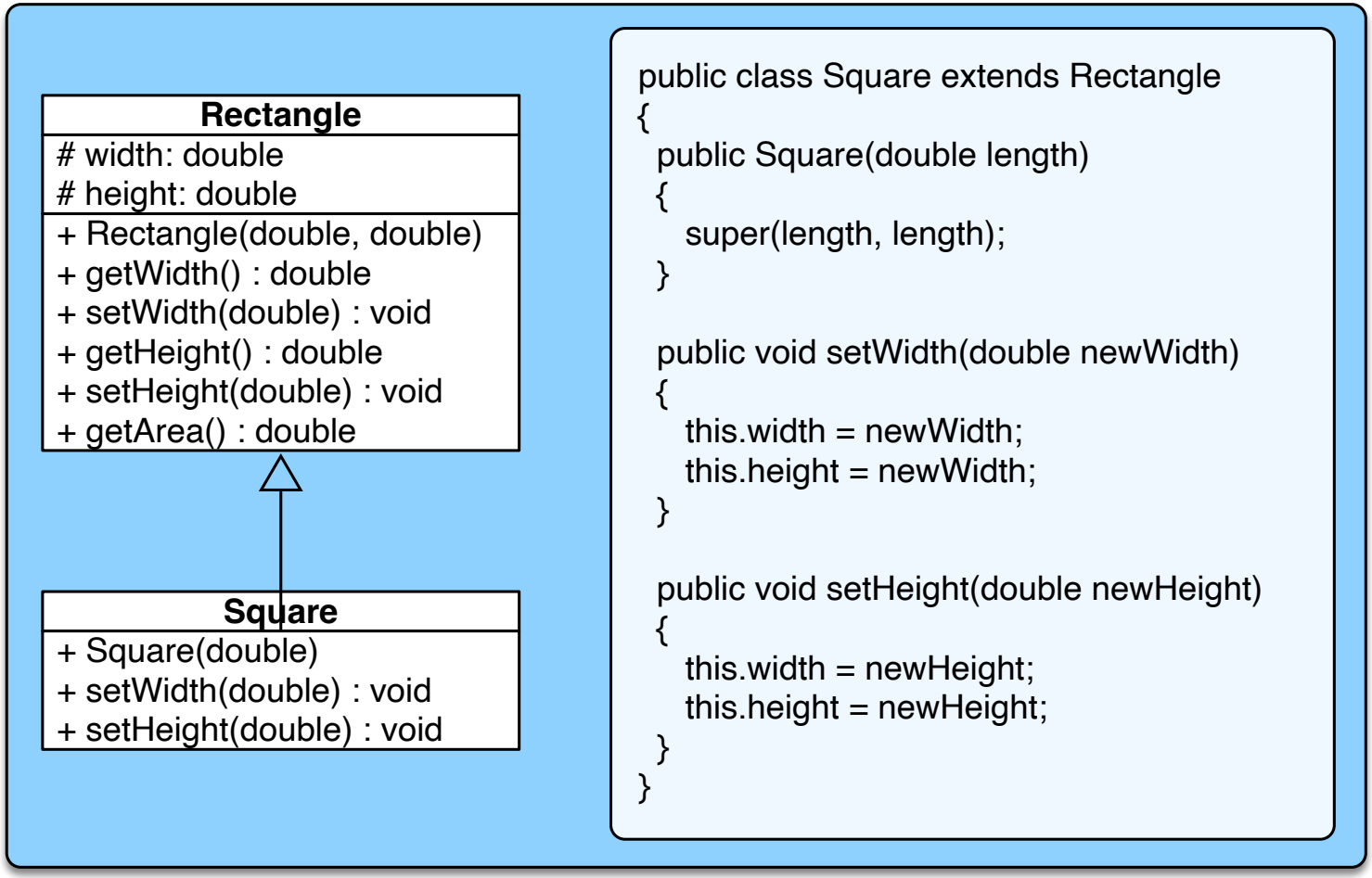
- Dans un programme, un objet doit pouvoir être **remplacé par un objet d'un sous-type** sans en altérer le fonctionnement



IUT Valence Université Grenoble Alpes

Une mauvaise conception

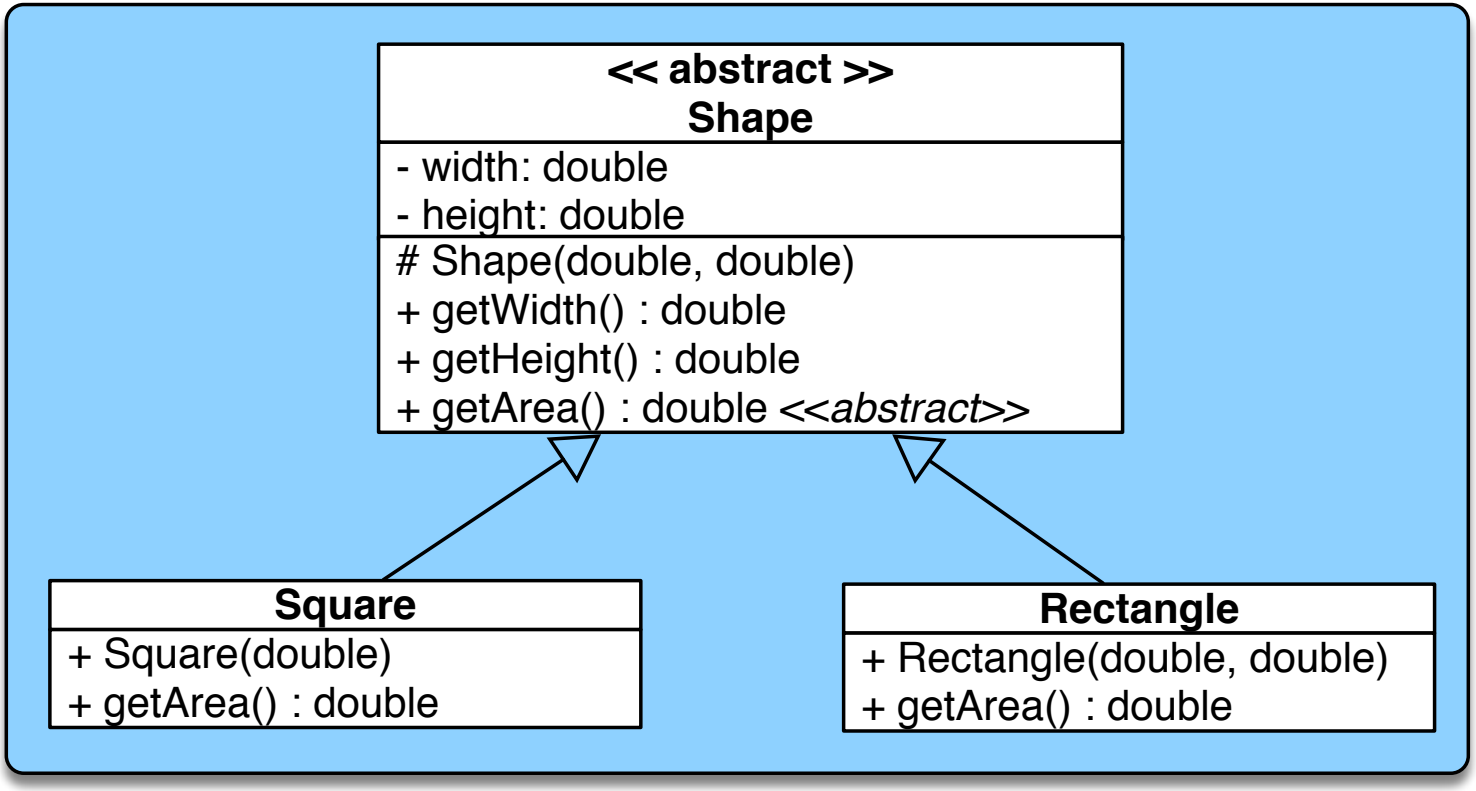
- Pourquoi ?



- *Exemple inspiré de*

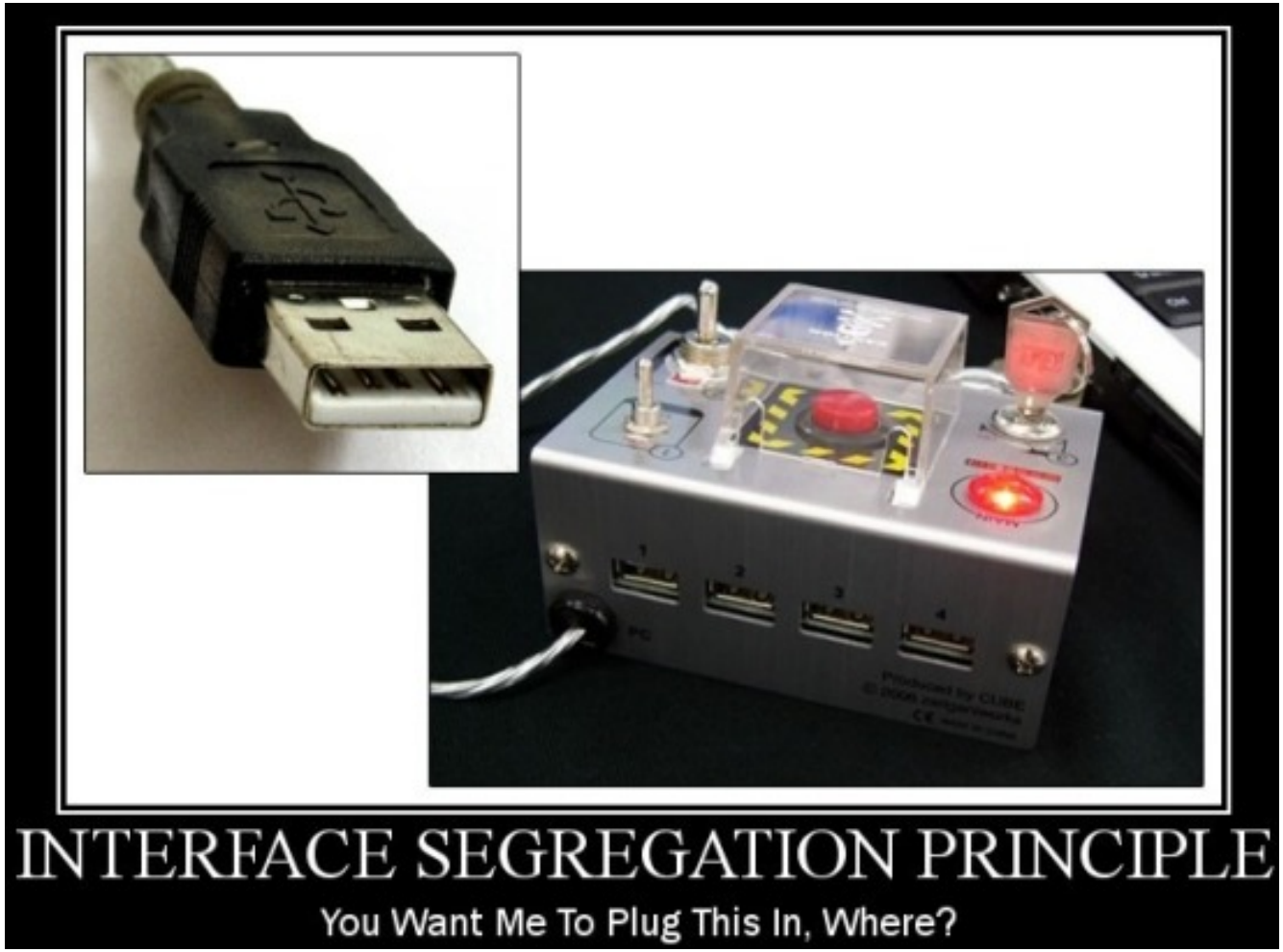
http://www.oodesign.com/liskov-s-substitution-principle.html

Une meilleure conception



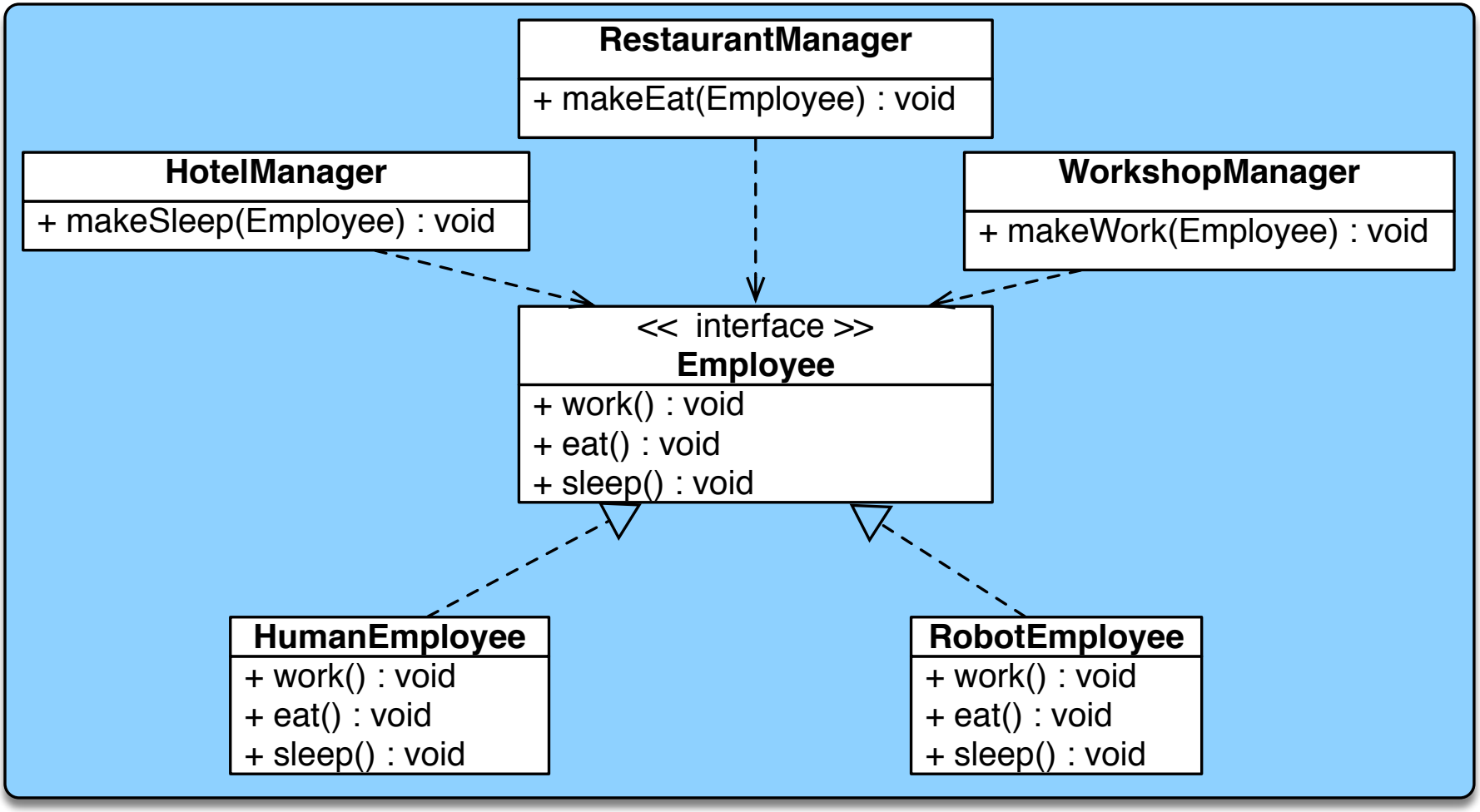
Interface Segregation Principle

- Un client **ne doit pas dépendre de méthodes qu'il n'utilise pas**



Une mauvaise conception

- Pourquoi ?

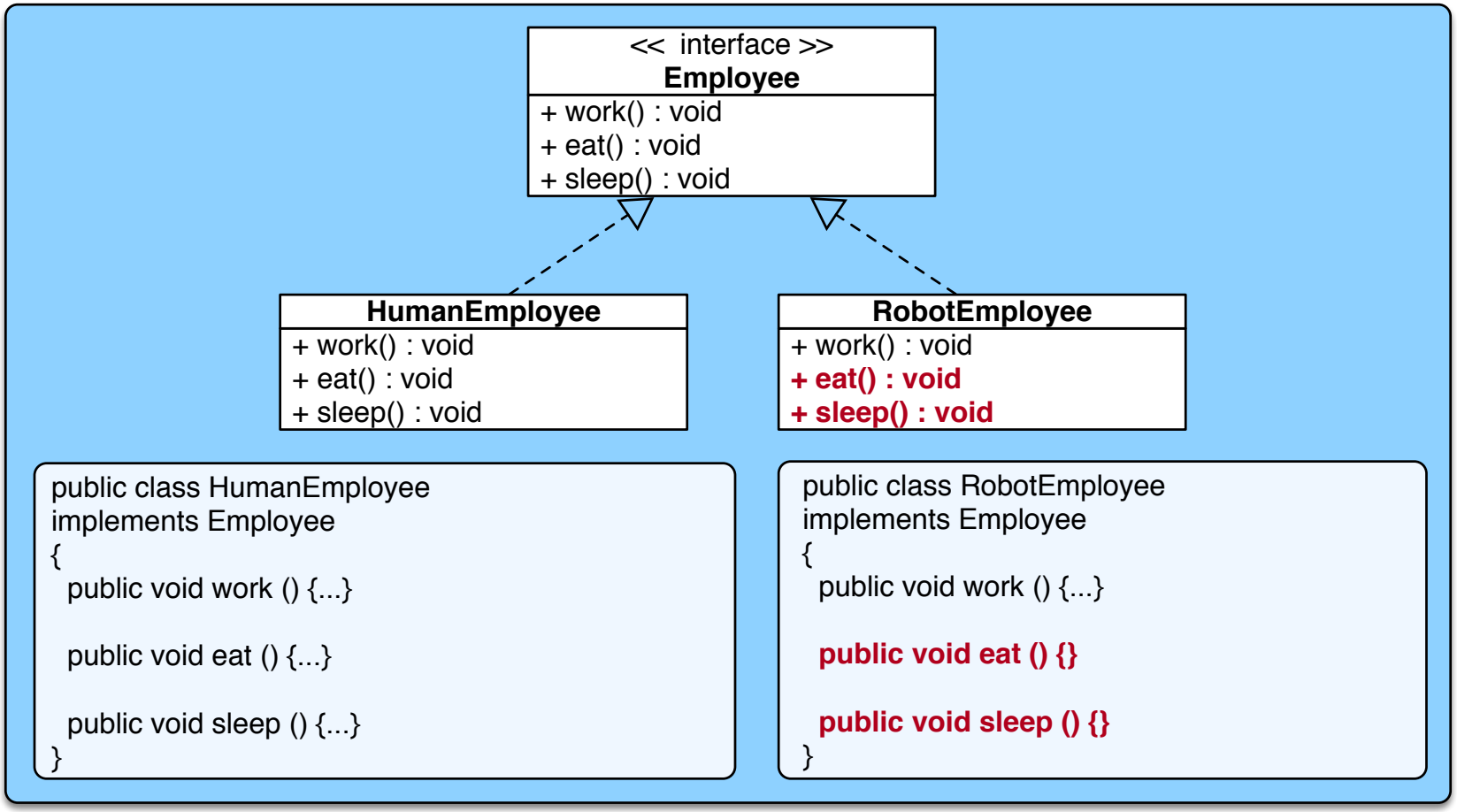


- *Exemple inspiré de*

http://www.oodesign.com/interface-segregation-principle.html

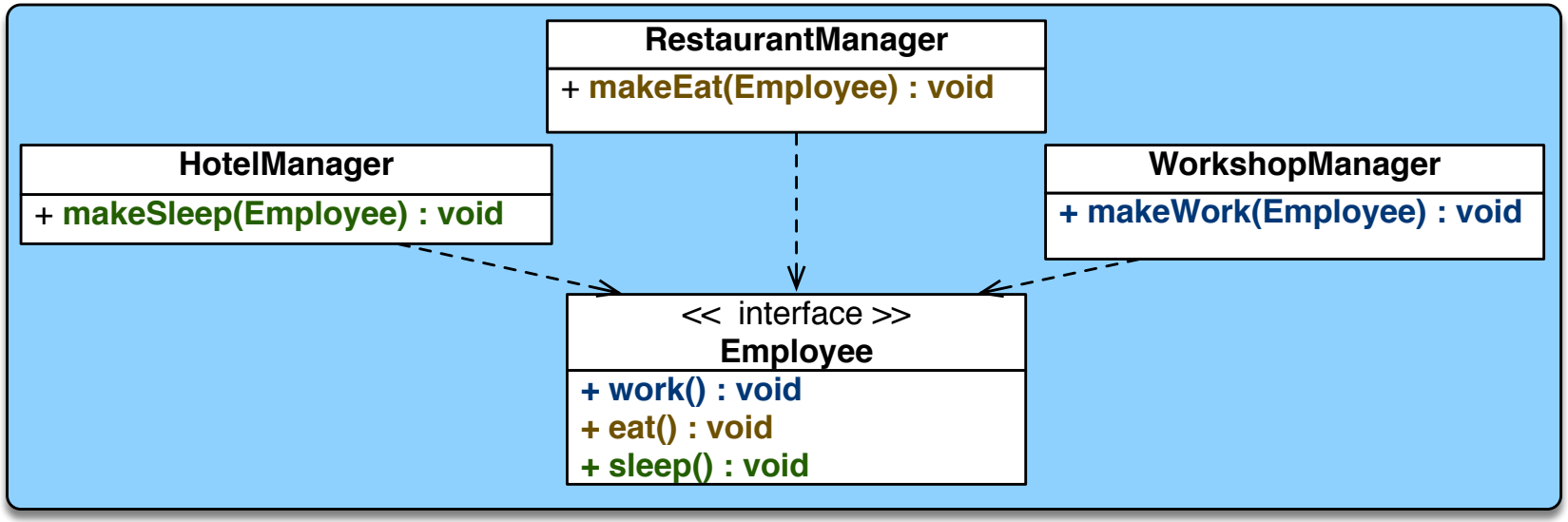
Une mauvaise conception

- L'interface est « surdimensionnée » pour les « fournisseurs »

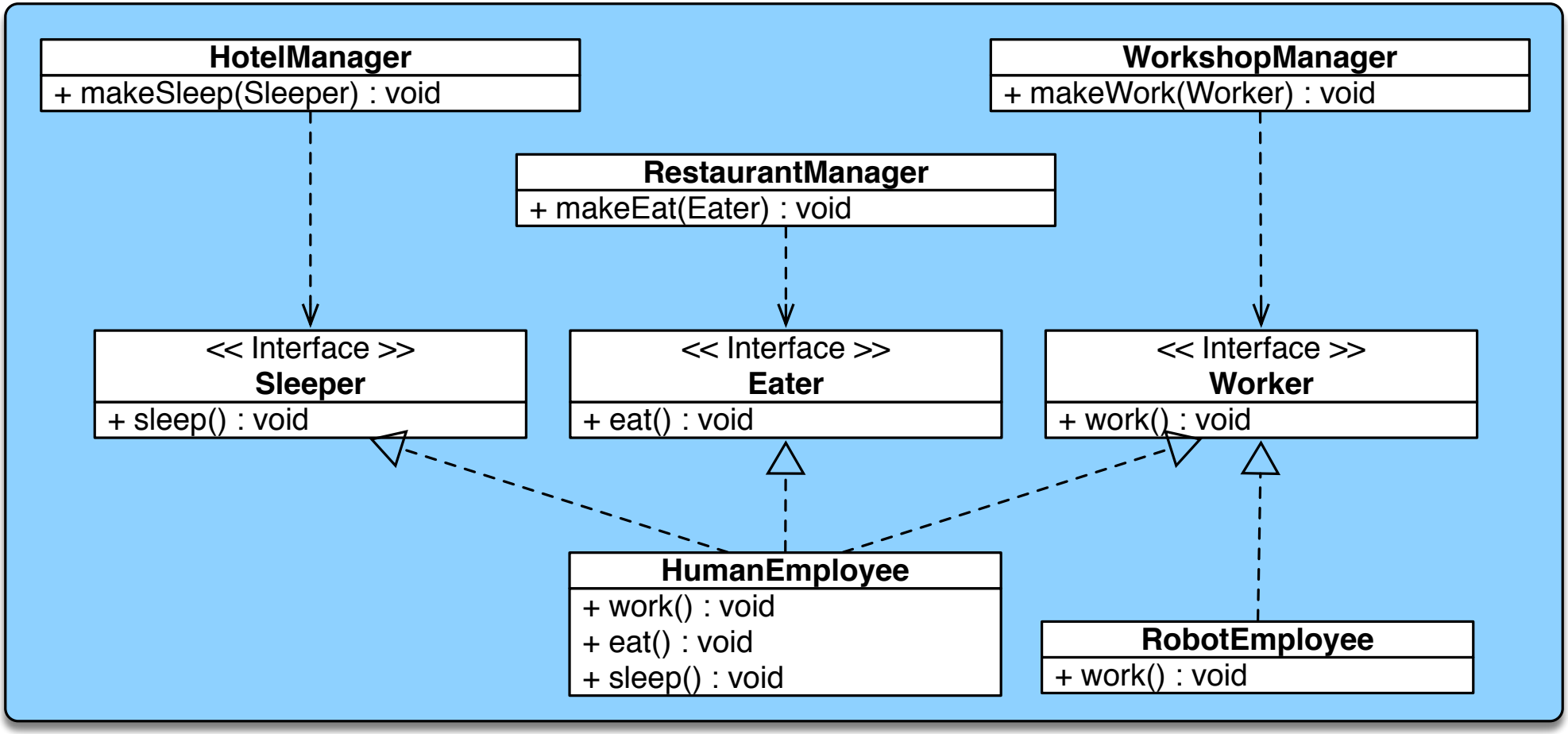


Une mauvaise conception

- L'interface est « surdimensionnée » pour les « consommateurs »

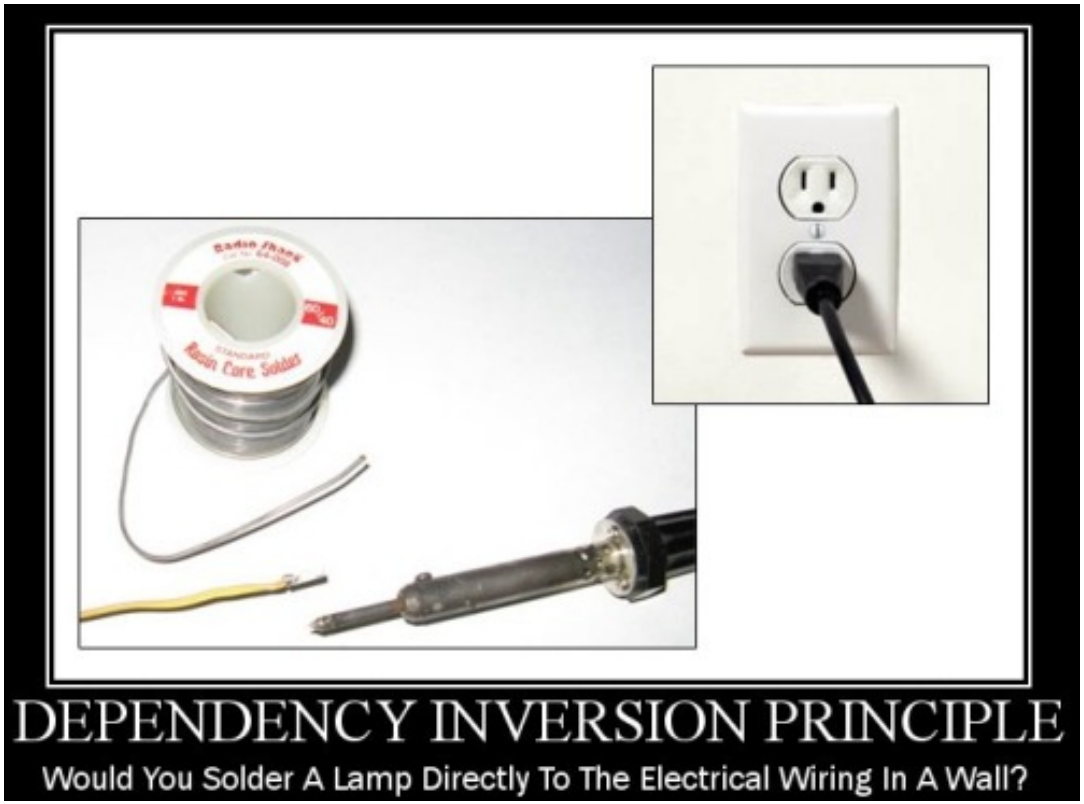


Une meilleure conception



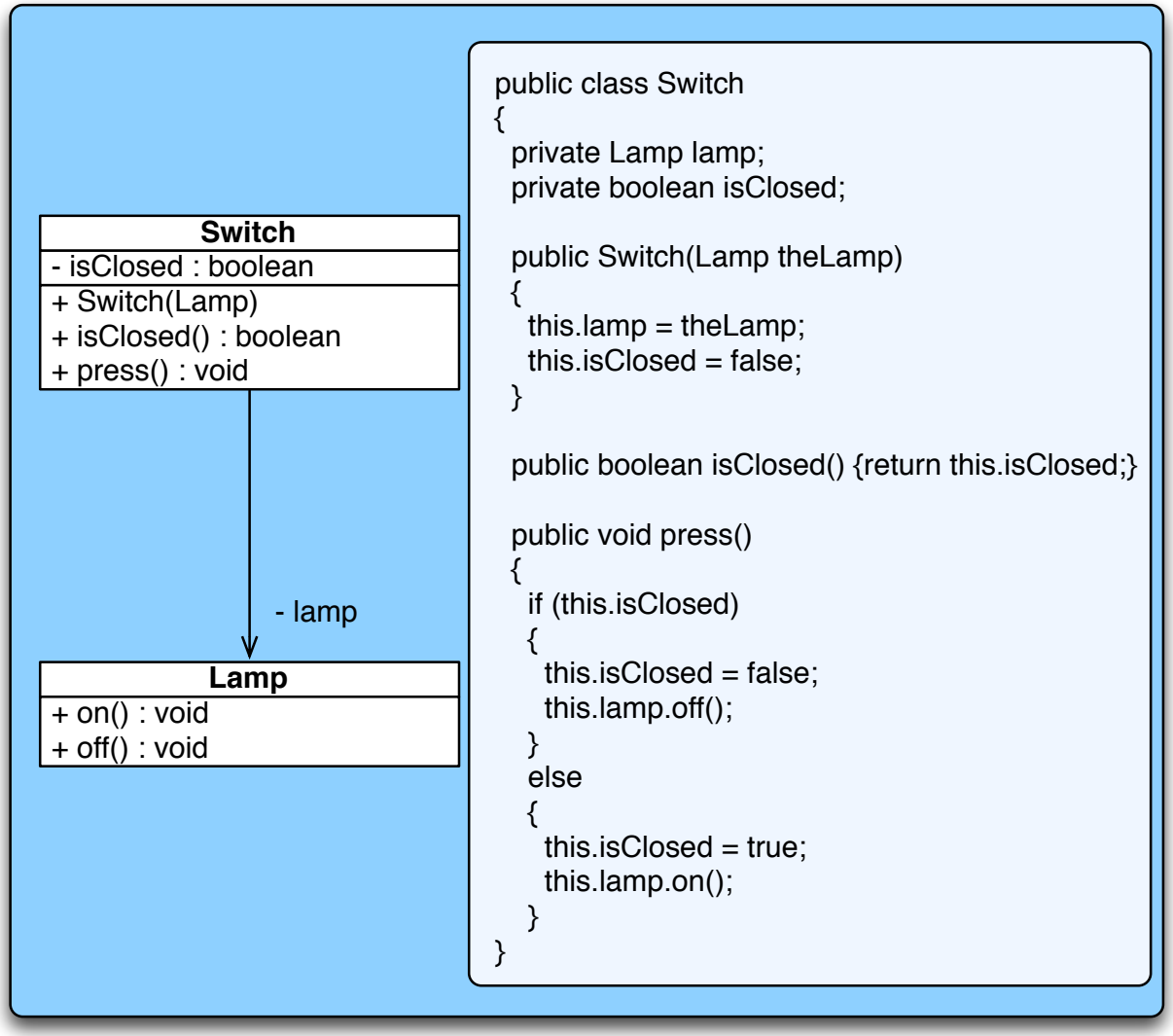
Dependency Inversion Principle

- Les **modules de haut-niveau ne doivent pas dépendre de modules de bas-niveau**. Les deux doivent dépendre d'abstractions
- Les **abstractions ne doivent pas dépendre des détails**, les **détails doivent dépendre des abstractions**



IUT Valence Université Grenoble Alpes

Une mauvaise conception

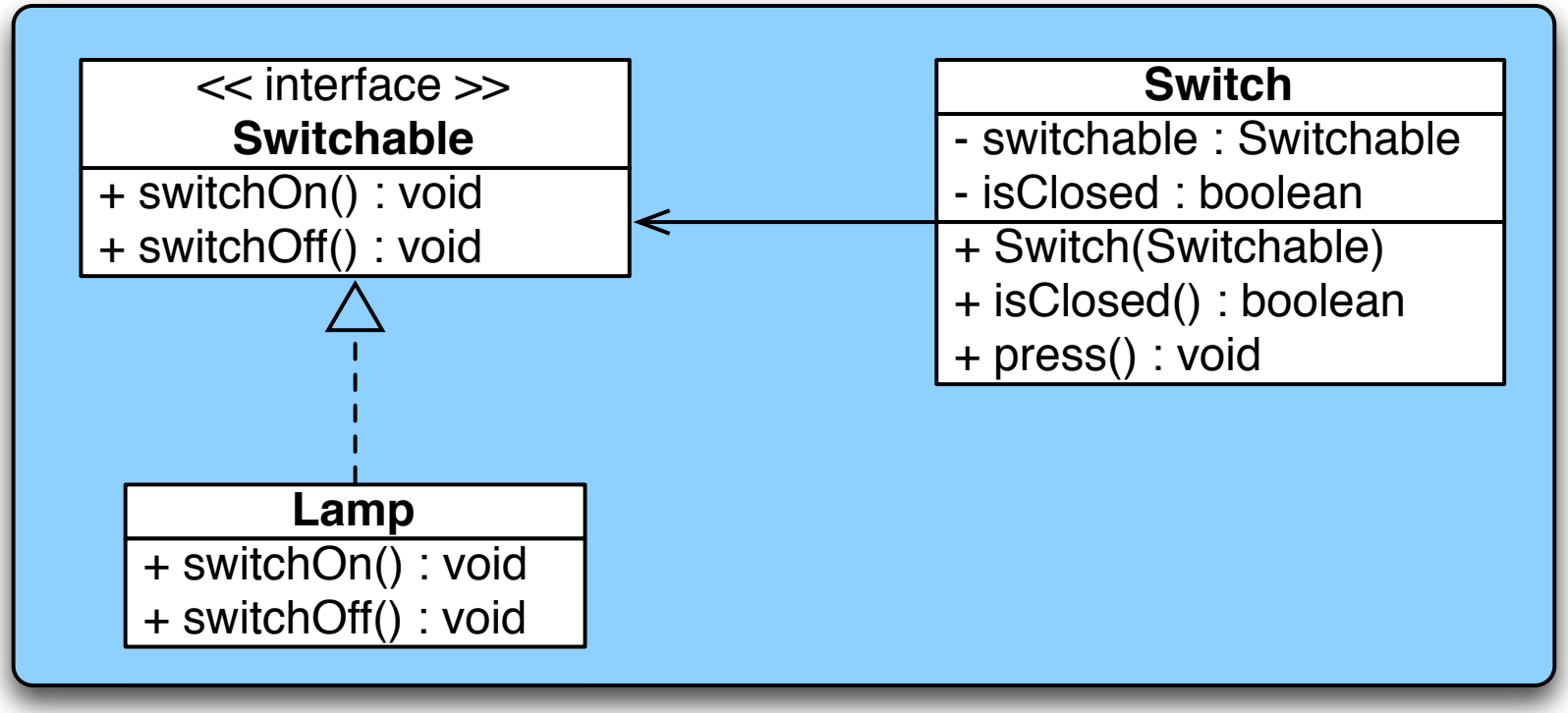


- *Exemple inspiré de*

http://kevin-buchanan.tumblr.com/post/74023212483/dependency-inversion-principle

IUT Valence Université Grenoble Alpes

Une meilleure conception



IUT Valence Université Grenoble Alpes

Loi de Demeter

- Une méthode `f` d'une classe `C` ne devrait appeler que des méthodes
  - de `C` ou d'un objet défini comme attribut de `C`
  - d'un objet créé par `f` ou passé en paramètre de `f`

DRY / DIE

- **D**on't **R**epeat **Y**ourself $\leftrightarrow$ **D**uplication **I**s **E**vil

Citation (extraite de *The Pragmatic Programmer)*

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Andy Hunt / Dave Thomas

IUT Valence Université Grenoble Alpes

YAGNI

Citation

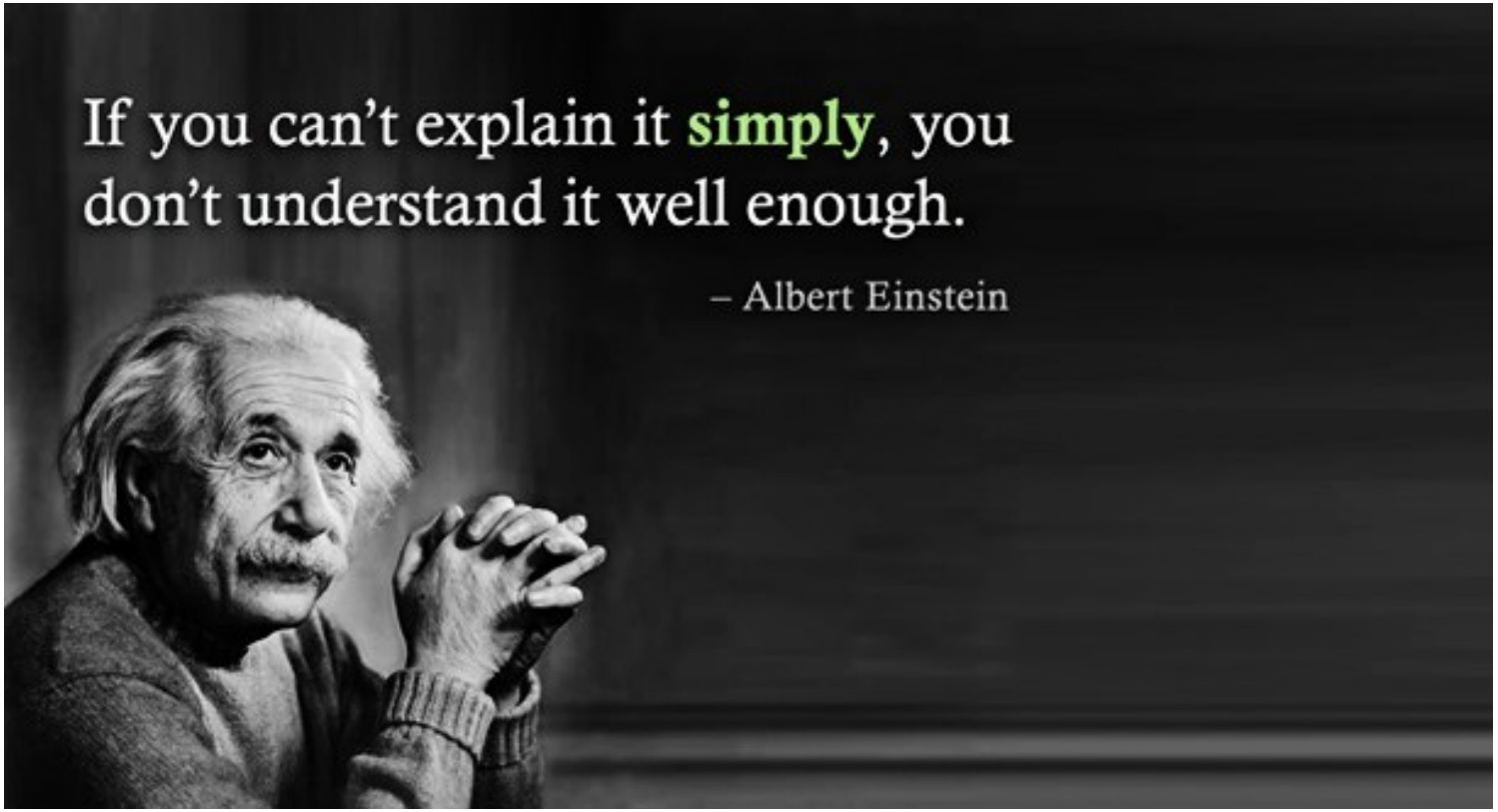
Always implement things when you actually need them, never when you just foresee that you need them.

Ron Jeffries

IUT Valence Université Grenoble Alpes

KISS

- Keep It Simple, Stupid



IUT Valence Université Grenoble Alpes

Fin !

IUT Valence
Université Grenoble Alpes