SOLID

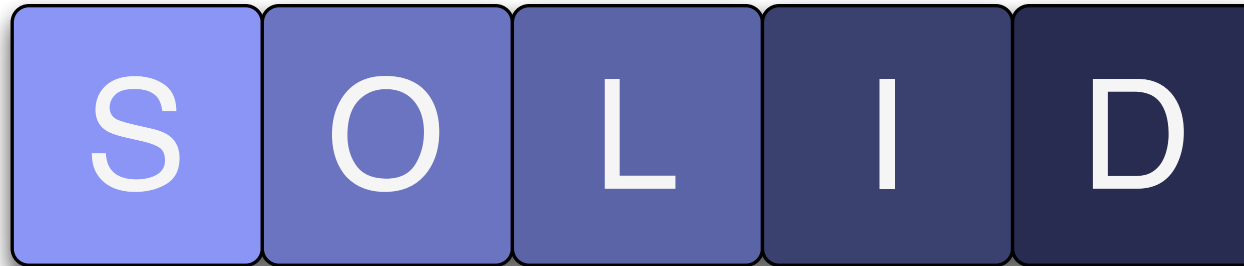
Sébastien Jean

IUT de Valence
Département Informatique

v3.1, 16 octobre 2024

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**

Une mauvaise conception ?

- Combien de responsabilités possède la classe ou sa méthode ?

CalculatingMachine
- result : int
+ processAdd(int, int) : void

```
public class CalculatingMachine
{
  private int result;

  public void processAdd(int a, int b)
  {
    this.result = a+b;
    System.out.println(this.result);
  }
}
```

- *Exemple inspiré de*

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

IUTA Valence
Université Grenoble Alpes

Une mauvaise conception

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

CalculatingMachine
- result : int
+ processAdd(int, int) : void

```
public class CalculatingMachine
{
  private int result;

  public void processAdd(int a, int b)
  {
    this.result = a+b;                    // resp. 1

    System.out.println(this.result); // resp. 2
  }
}
```

IUTA Valence
Université Grenoble Alpes

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

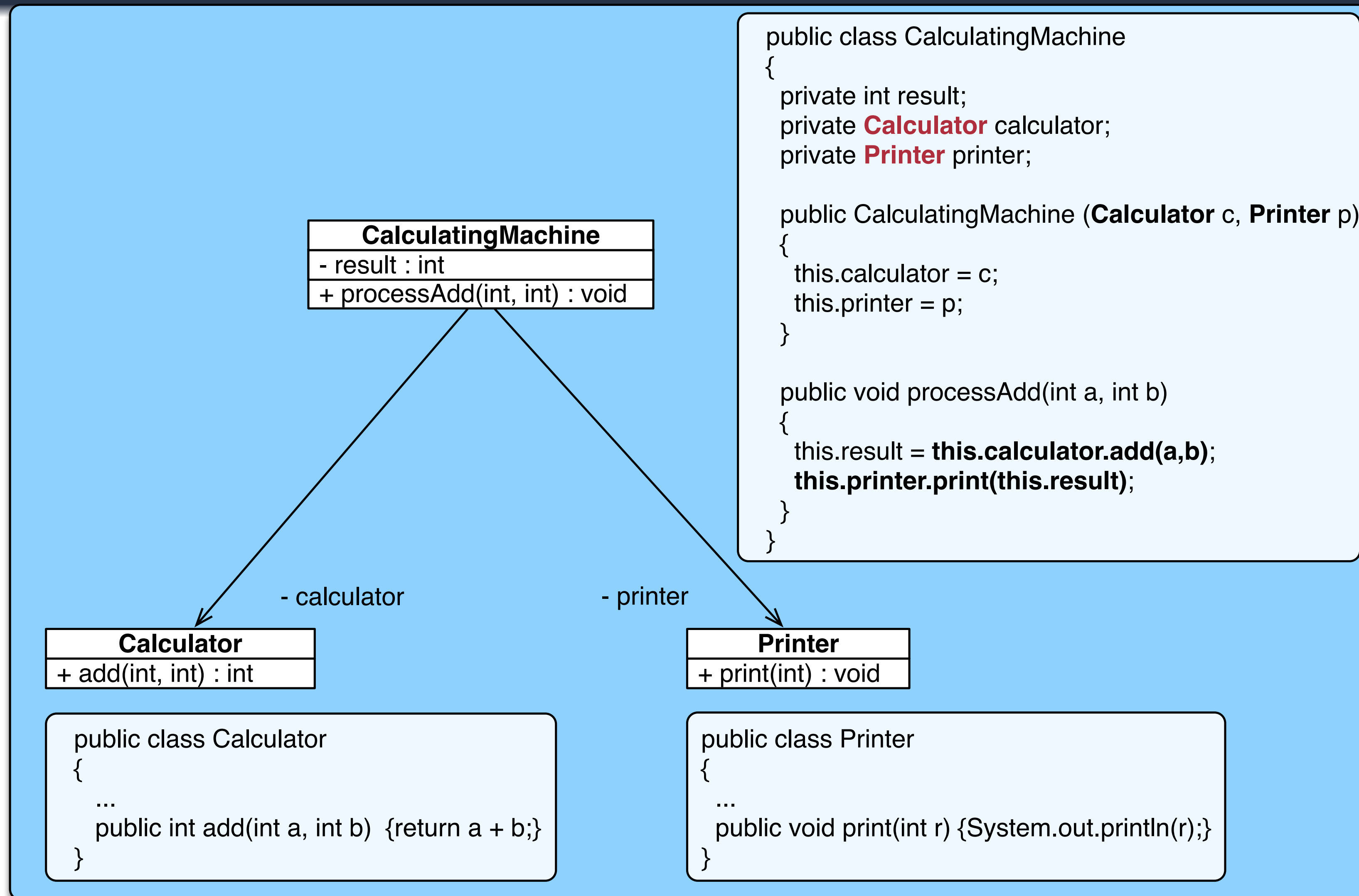
CalculatingMachine
- result : int
+ processAdd(int, int) : void **- add(int, int) : int** **- print(int) : void**

```
public class CalculatingMachine
{
  private int result;

  public void processAdd(int a, int b)
  {
    this.result = this.add(a, b);
    this.print(this.result);
  }

  private int add(int a, int b) {return a+b;}

  private void print(int r) {System.out.println(r);}
}
```

- Mais la classe a toujours les deux responsabilités !

IUTA Valence Université Grenoble Alpes

Une meilleure conception



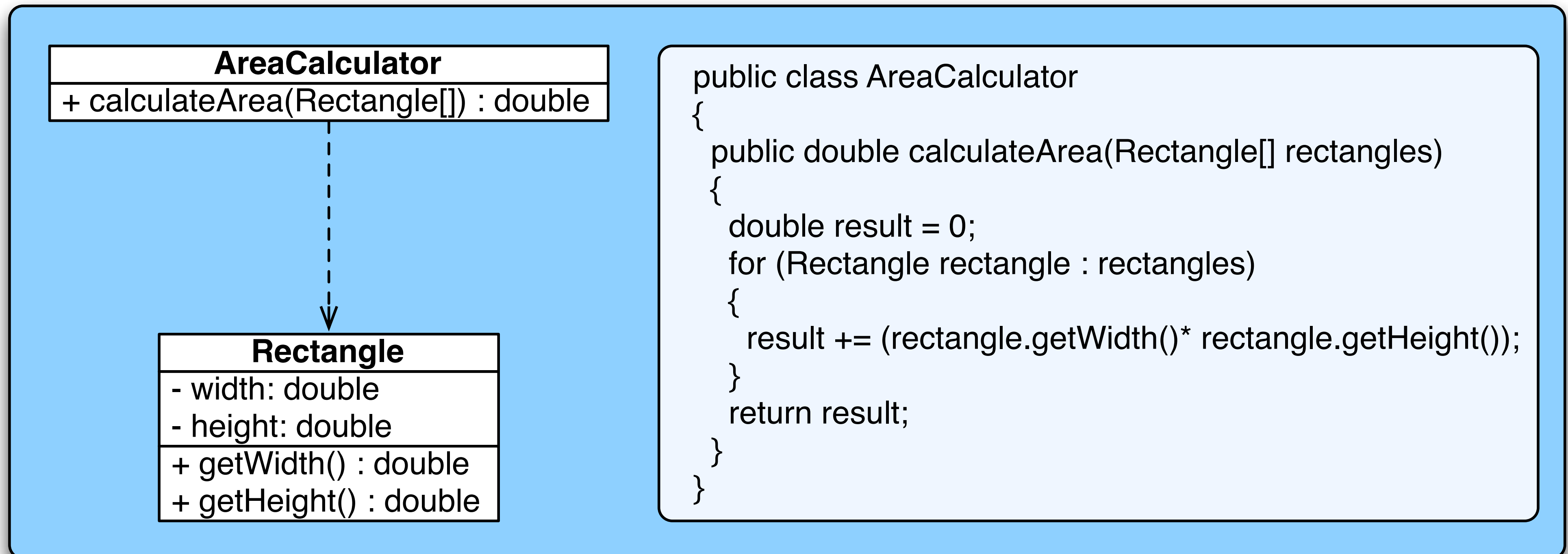
- La **classe et la méthode n'ont plus qu'une responsabilité**
- Et on pourrait encore aller plus loin . . .

Open/Closed Principle

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

IUTA Valence
Université Grenoble Alpes

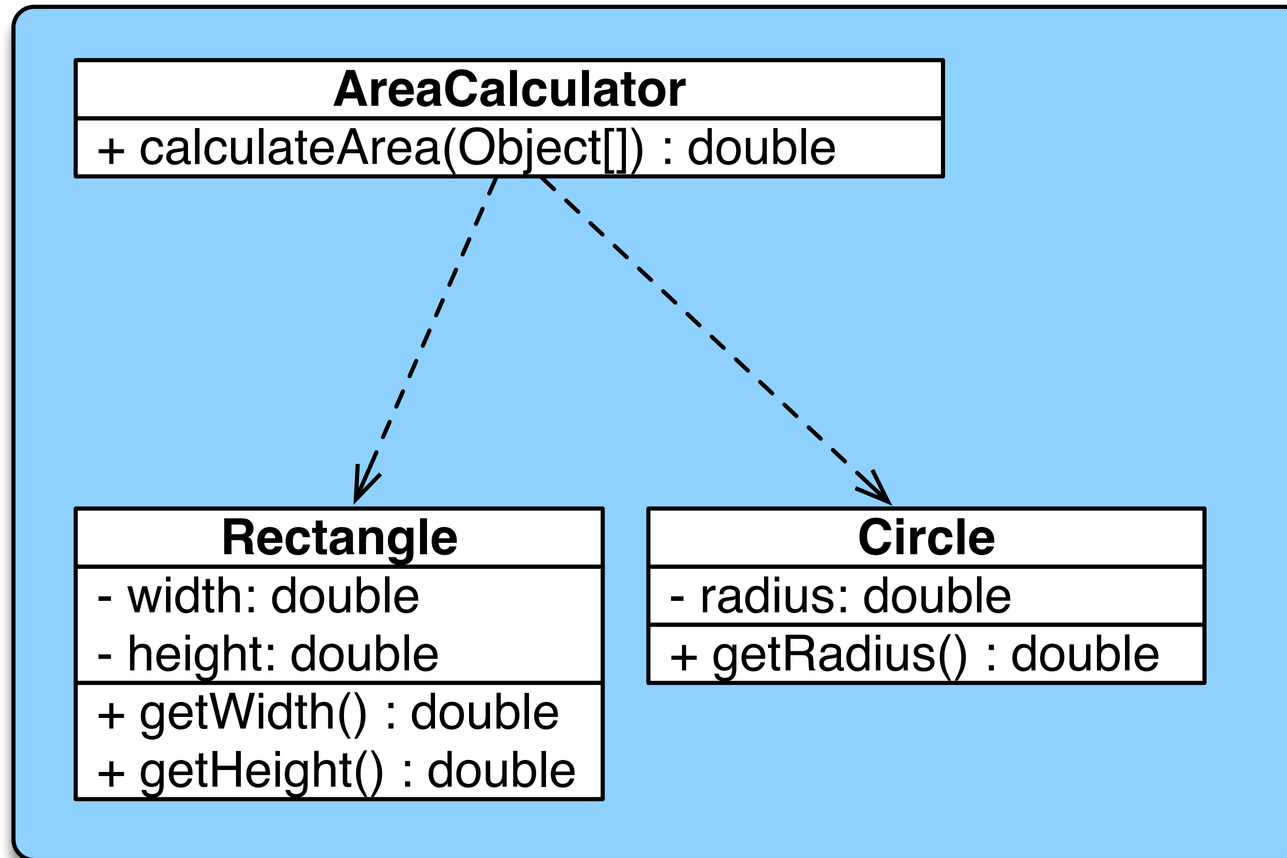
Exemple



- *Exemple inspiré de*

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

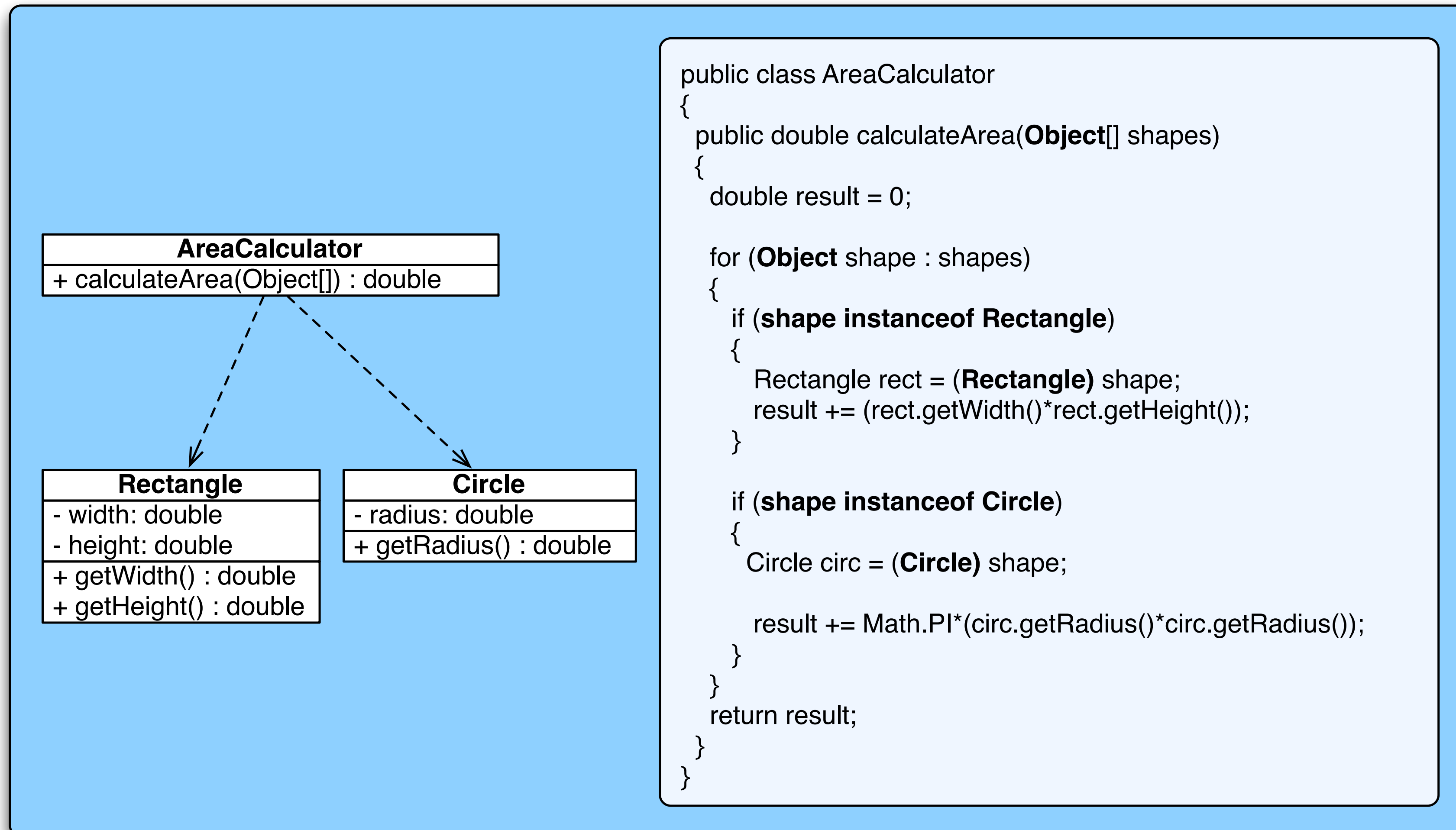
Prise en compte de formes différentes : une solution simple



- Pourquoi est-ce une mauvaise idée ?

Prise en compte de formes différentes : une solution simple

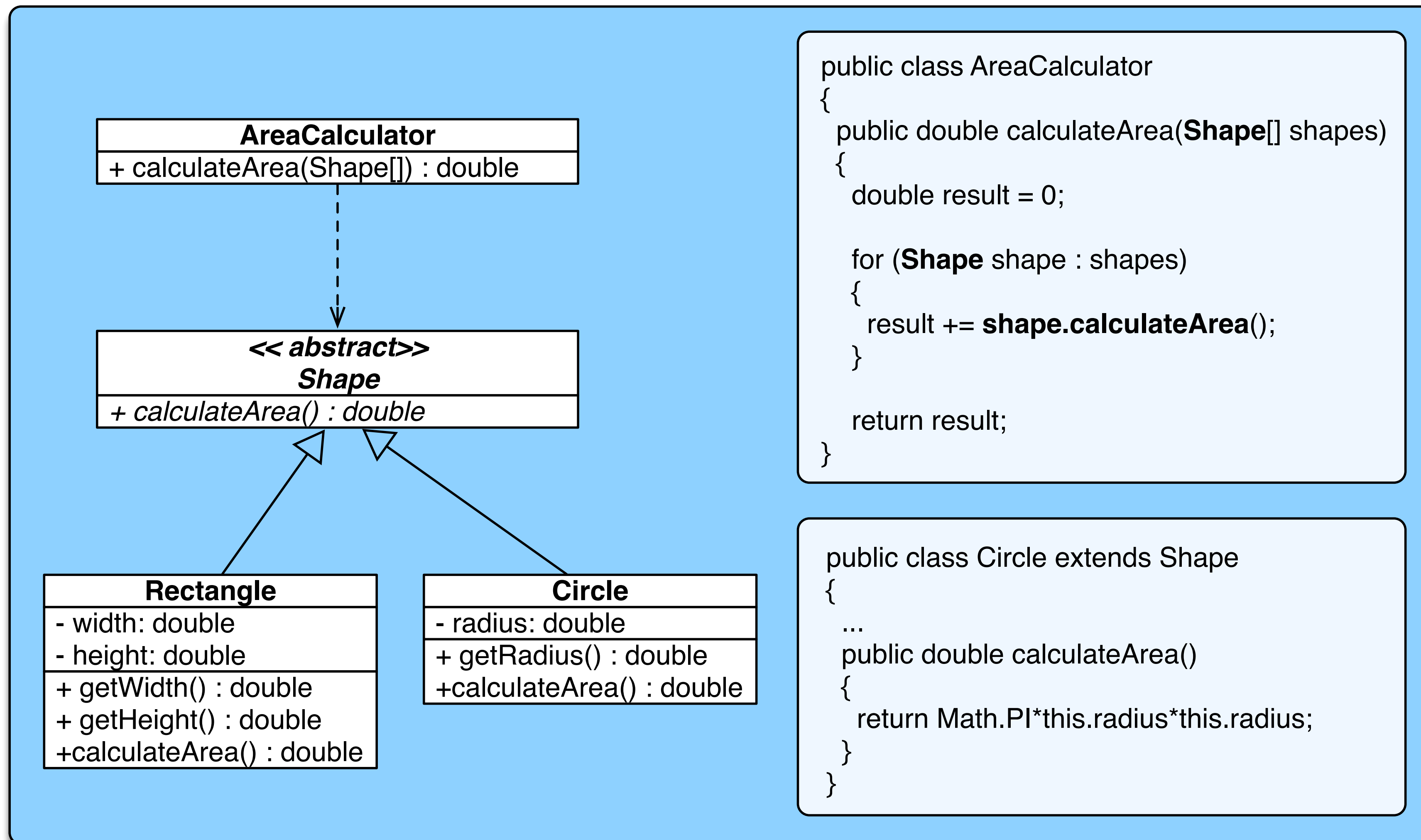
- La généricité apportée par l'utilisation d'`Object` nécessite de **modifier le code de la classe** pour la prise en compte de nouvelles formes



IUTA Valence Université Grenoble Alpes

Une meilleure conception

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



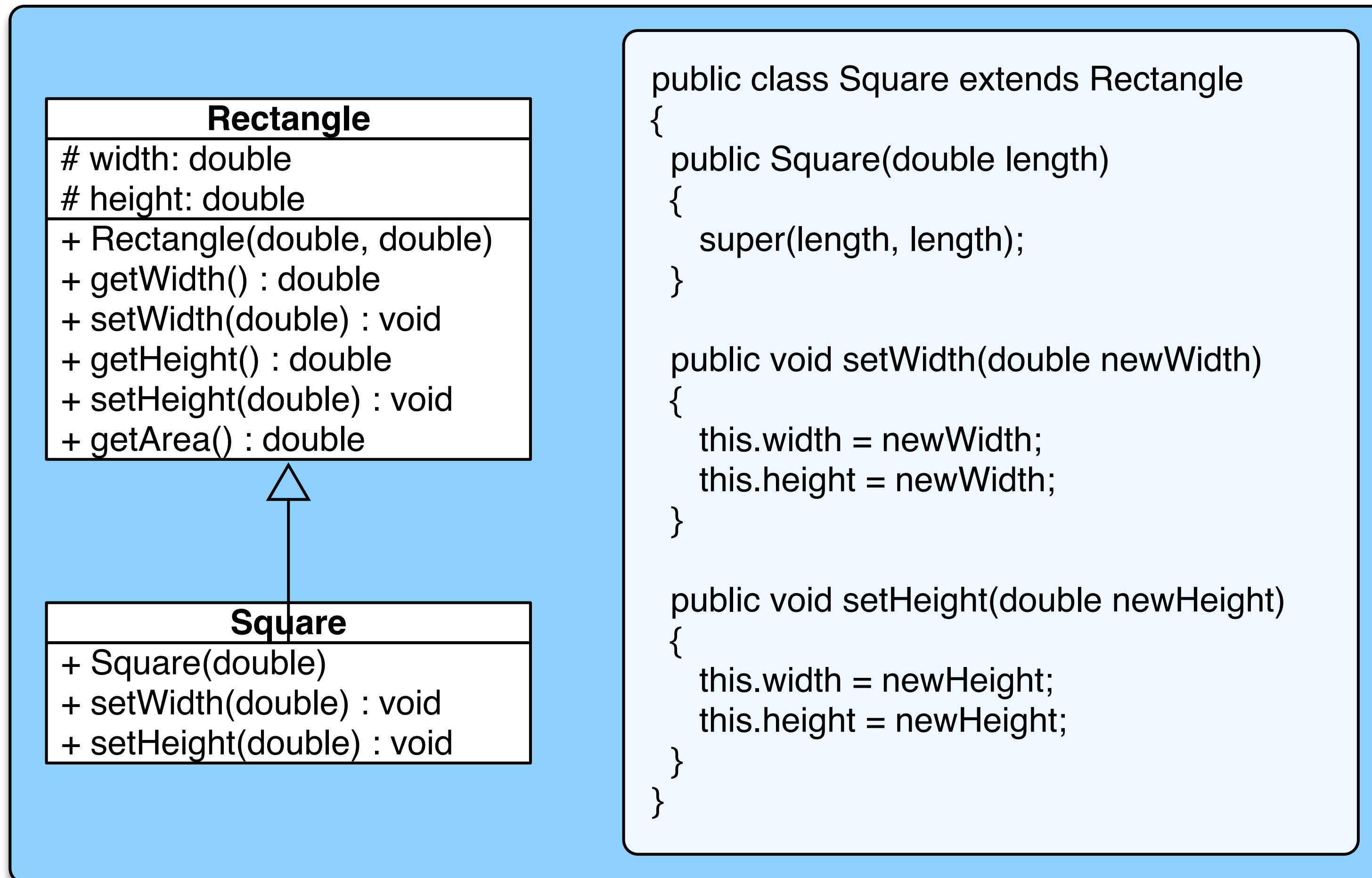
Liskov Substitution Principle

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

IUTA Valence Université Grenoble Alpes

Une mauvaise conception

- Pourquoi ?

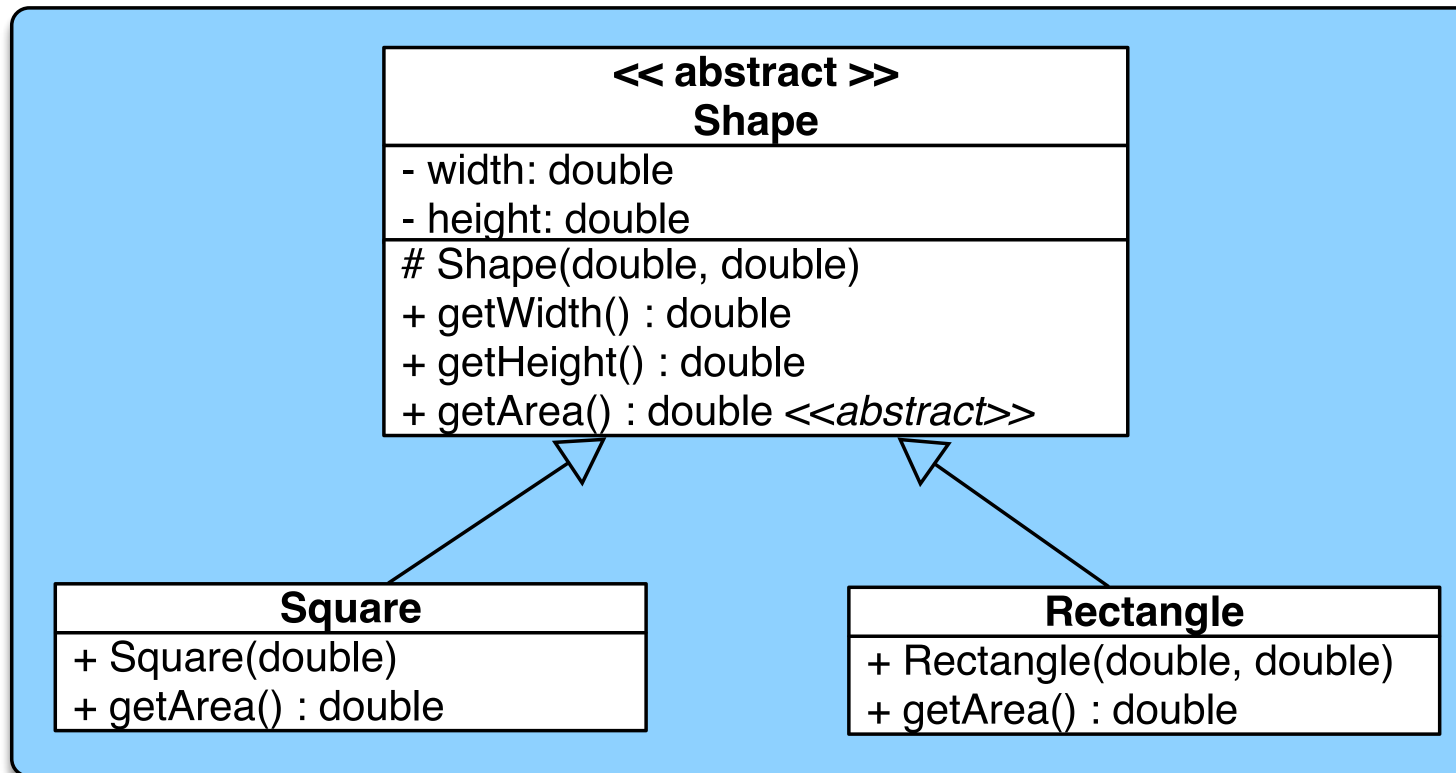


- *Exemple inspiré de*

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

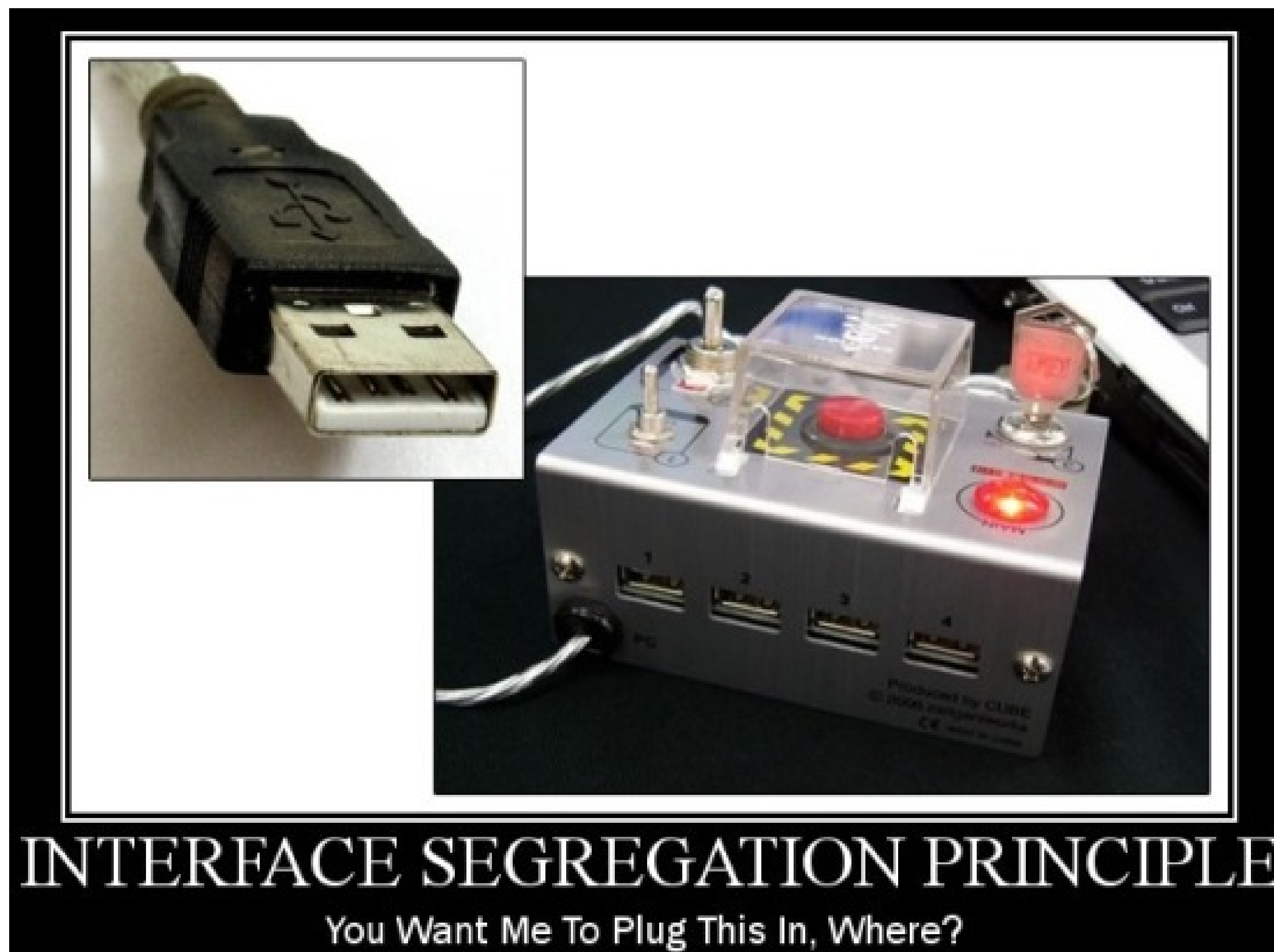
IUT Valence Université Grenoble Alpes

Une meilleure conception



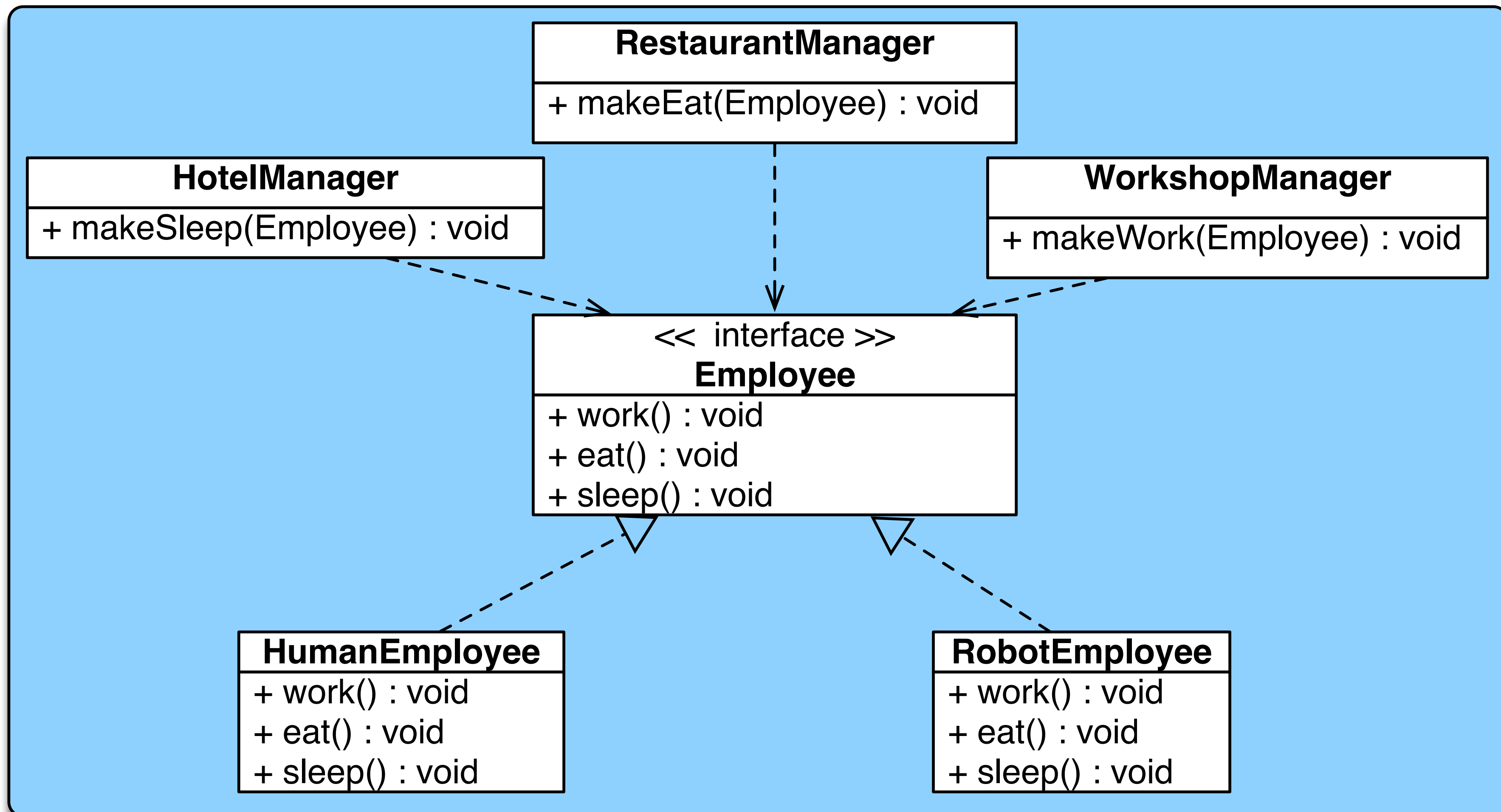
Interface Segregation Principle

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



Une mauvaise conception

- Pourquoi ?



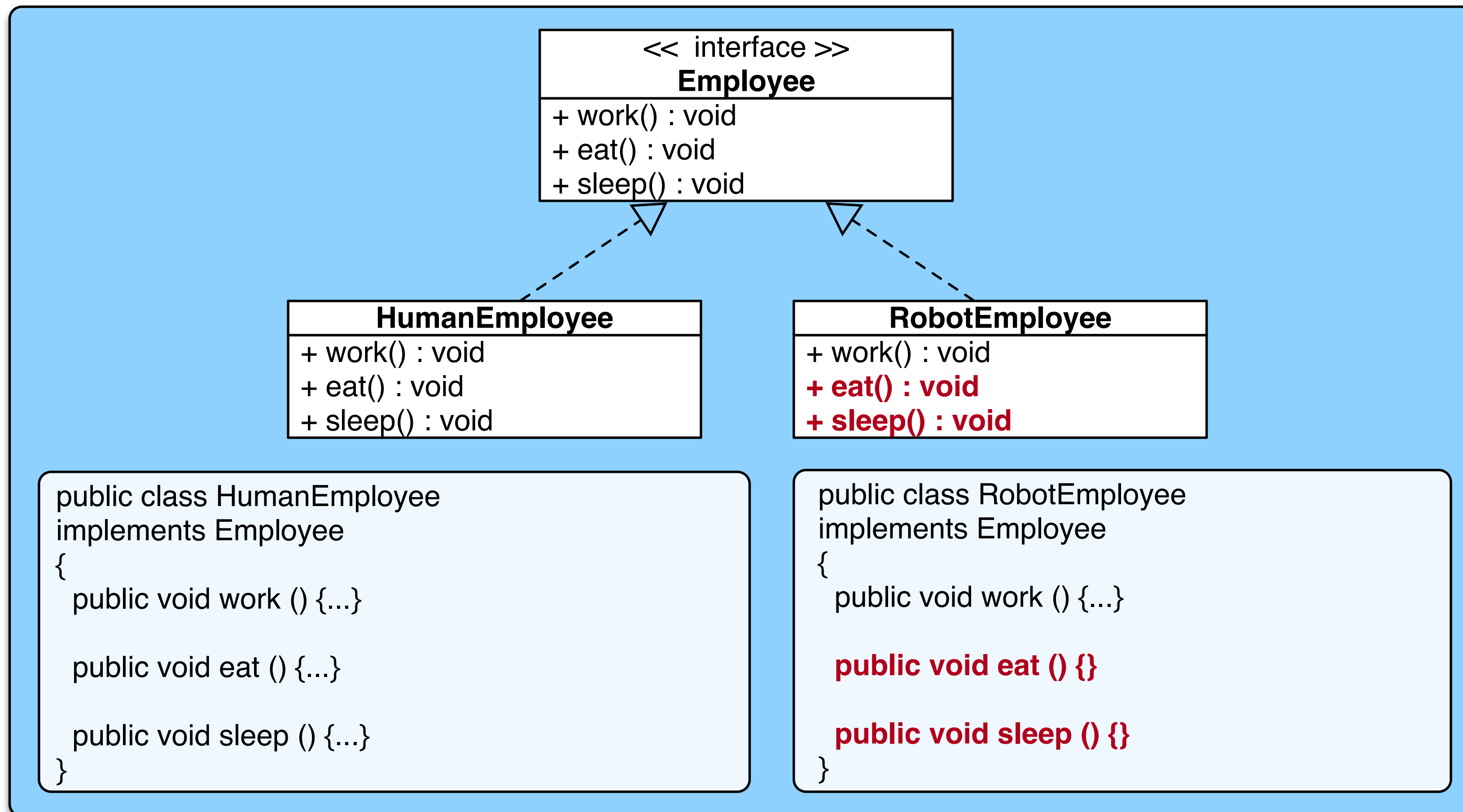
- *Exemple inspiré de*

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

IUTA Valence Université Grenoble Alpes

Une mauvaise conception

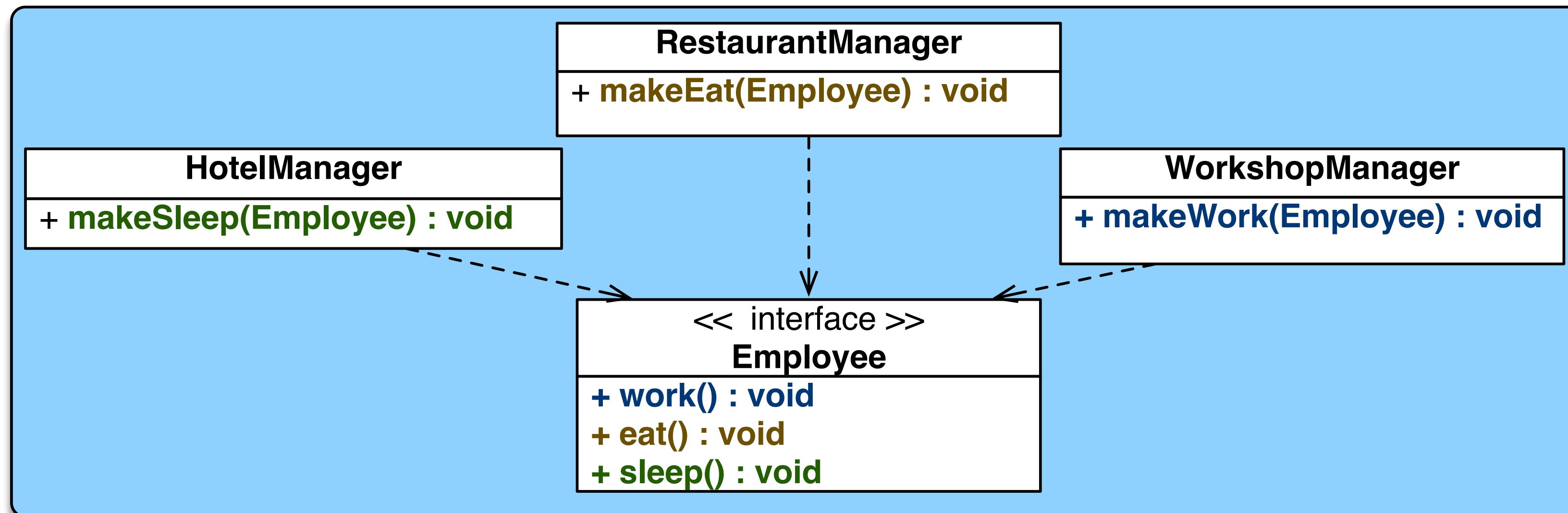
- L'interface est « surdimensionnée » pour les « fournisseurs »



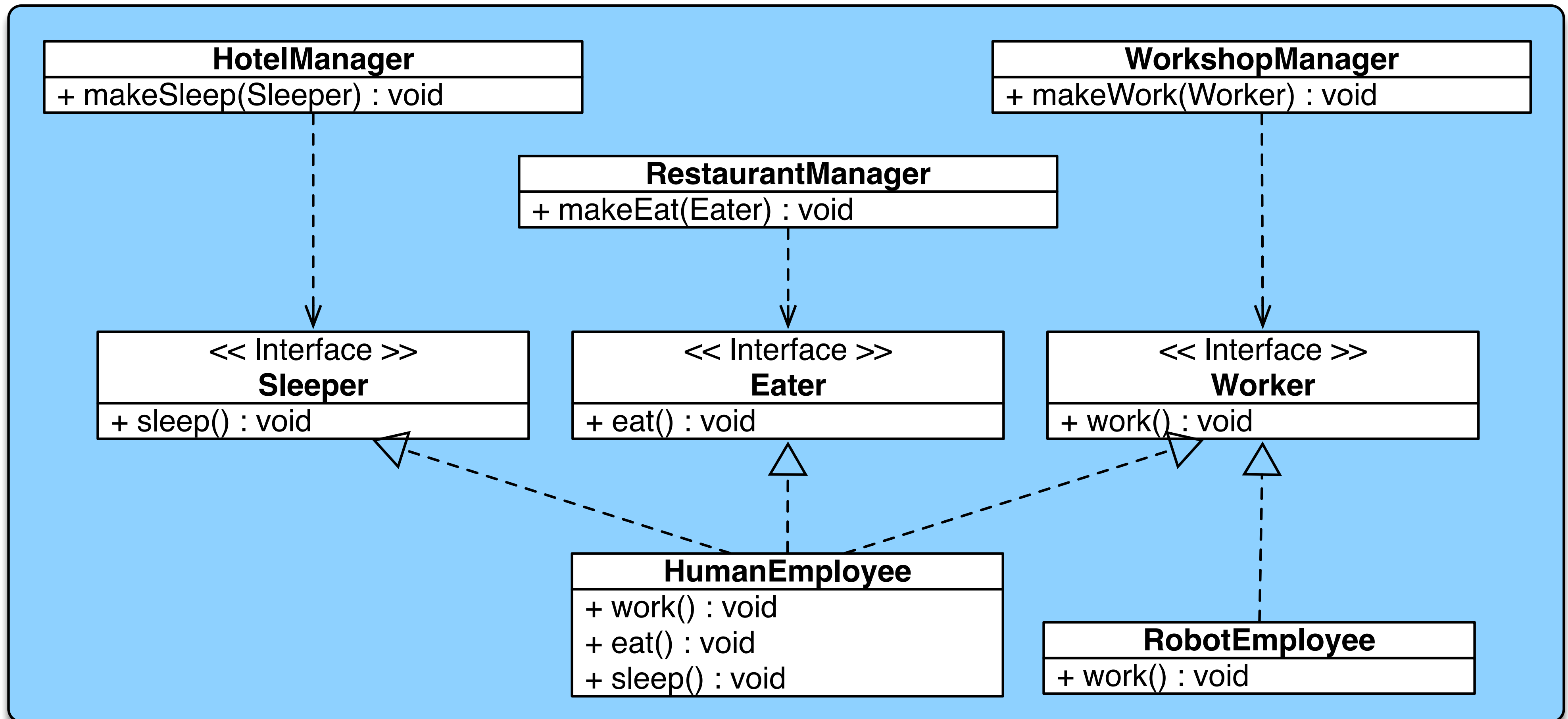
IUT Valence Université Grenoble Alpes

Une mauvaise conception

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



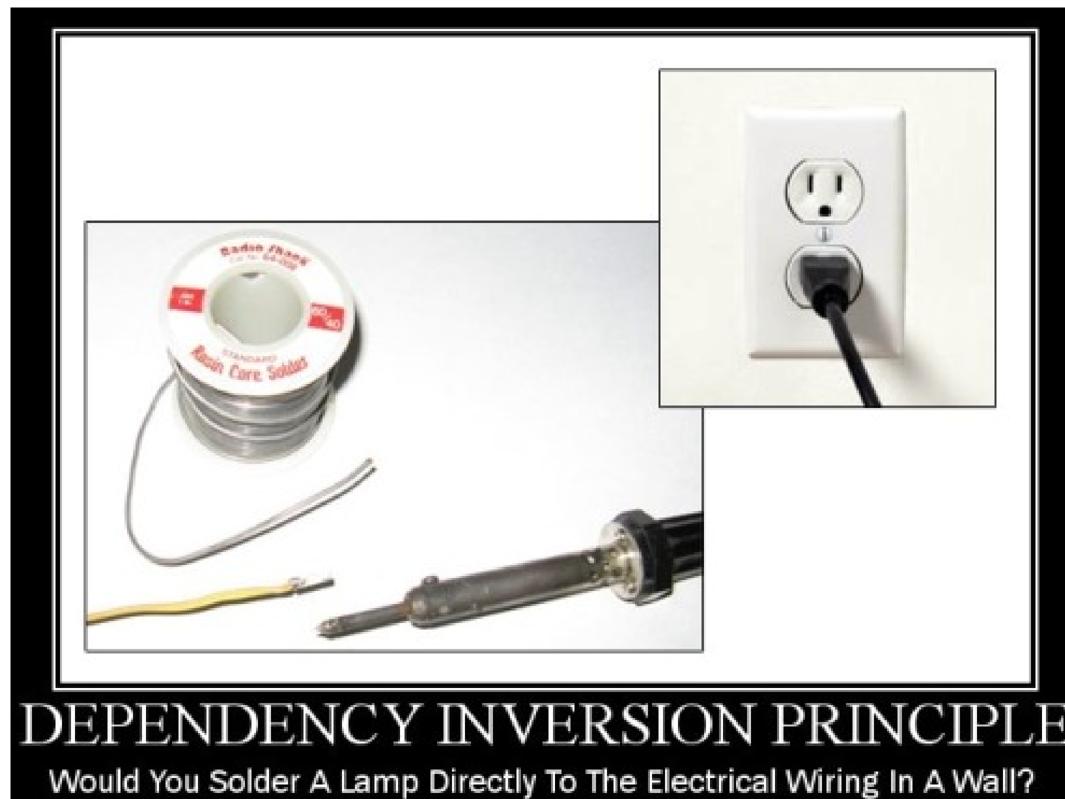
Une meilleure conception



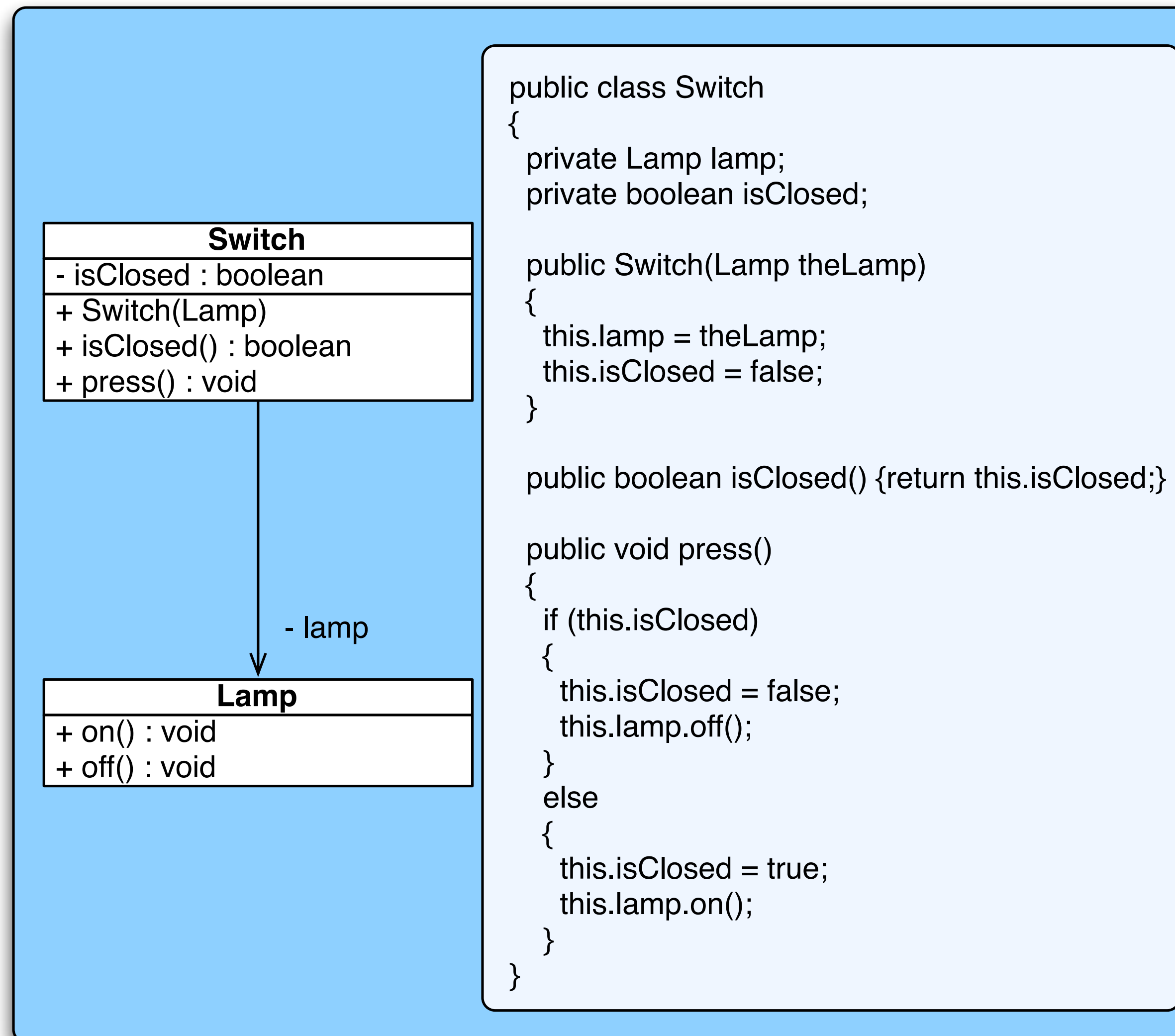
IUT Valence
Université Grenoble Alpes

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**



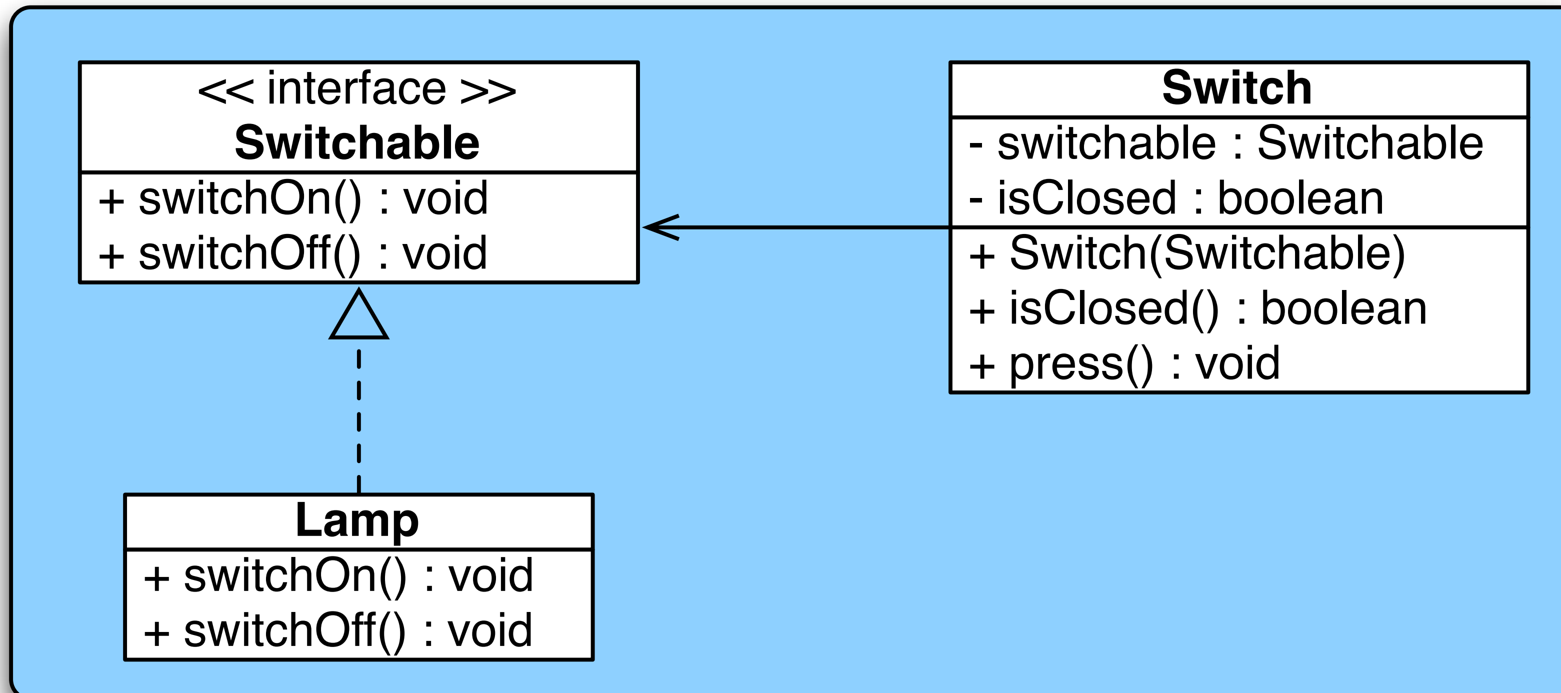
Une mauvaise conception



- *Exemple inspiré de*

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

Une meilleure conception



IUT Valence
Université Grenoble Alpes

Injection de dépendances

- L'**injection de dépendances** (*dependency injection*) permet d'implémenter le principe de l'inversion de contrôle

- **Création dynamique** des dépendances entre les différents objets
  - en s'appuyant sur une description (fichier de conf., métadonnées)
  - programmatiquement
- **Injection** de la référence dans l'objet destination
  - en passant un paramètre au constructeur
  - en appelant une méthode dédiée

IUT Valence
Université Grenoble Alpes

Nommage

Exercice

```
public class Game {

  public Game() {this.gameBoard = new Board(8, 8);}

  public void start() {
    boolean gameIsOver = false;
    Scanner in = new Scanner(System.in);
    while (!gameIsOver) {
      System.out.println(this.gameBoard.toString());
      Position position = null;
      while (true) {
        position = Position.parsePosition(in.readLine());
        if (this.gameBoard.isPositionValid(position)) break;
      }
      this.gameBoard.toogleAround(position);
      gameIsOver = this.gameBoard.isWon();
    }
}
```

Fin !

IUTA Valence
Université Grenoble Alpes