Les principes SOLID

Yassamine Seladji

yassamine.seladji@gmail.com

8 février 2021

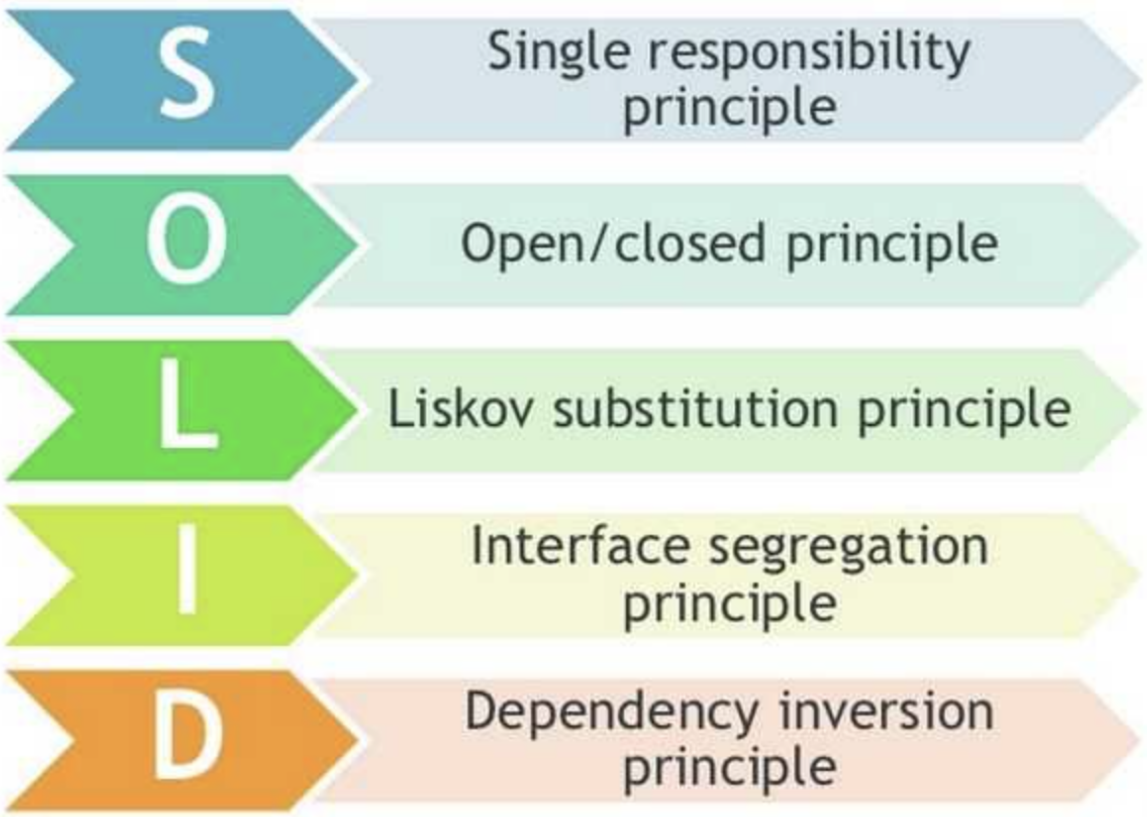
Le contexte

- La programmation orientée objet.
- Une conception de qualité.
  - Compréhensible.
  - Facile à maintenir.
  - Facile à étendre.

Le contexte

- Dans la conception orientée objet :
  - La classe : le block de construction principal.
  - Les relations entre classes : définissent la qualité de la conception.
  - Les bonnes pratiques de la programmation et de la conception.

Le principe SOLID



Le principe SOLID

- **S** **S**ingle responsability.
- **O** **O**pen/Close.
- **L** **L**iskov substitution.
- **I** **I**nterface segregation.
- **D** **D**ependency inversion.

Single responsibility

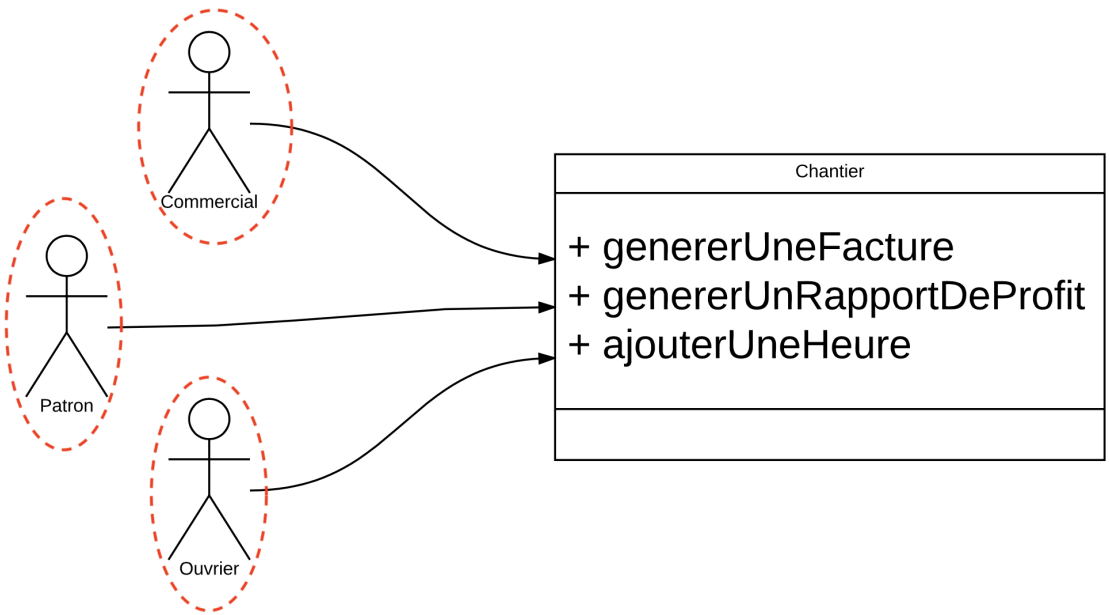
"A class should have only single responsibility"

Single responsibility

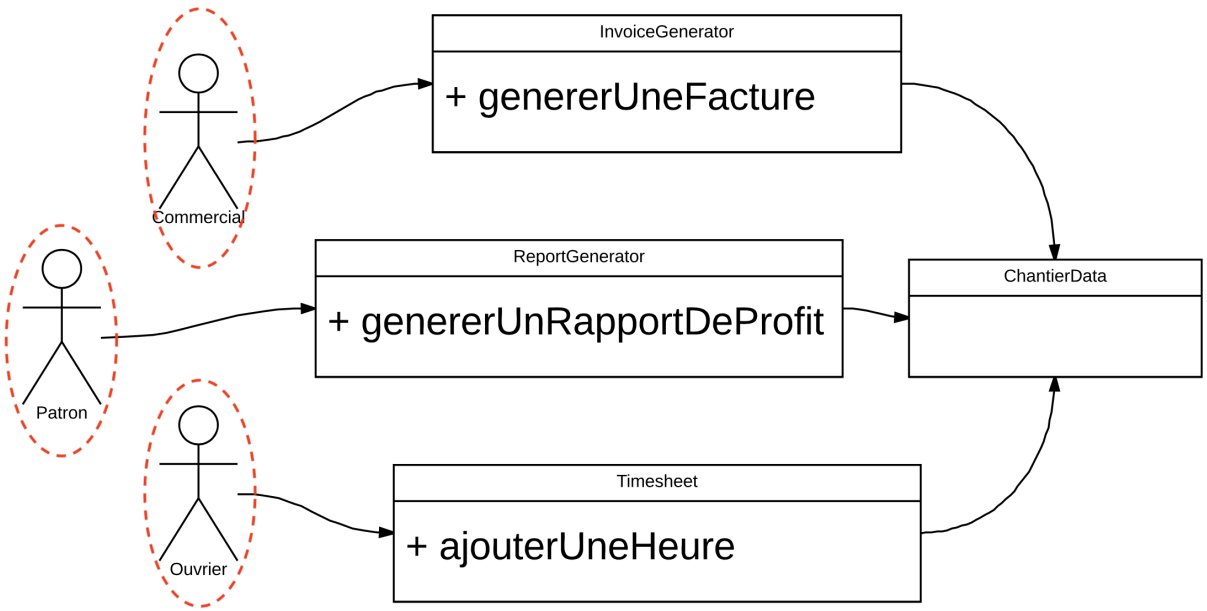
"A class should have only single responsibility"

- Le programmeur doit écrire, modifier et maintenir une classe avec une seul responsabilité.
- L'unique responsabilité de la classe est l'unique cause de changement.
- La bonne manière d'identifier les classes durant la conception.

Single responsibility



Single responsibility



Single responsibility

La responsabilité unique permet d'avoir :

- une conception faiblement couplée.
- des dépendances moins nombreuses et plus légères.

Le principe SOLID

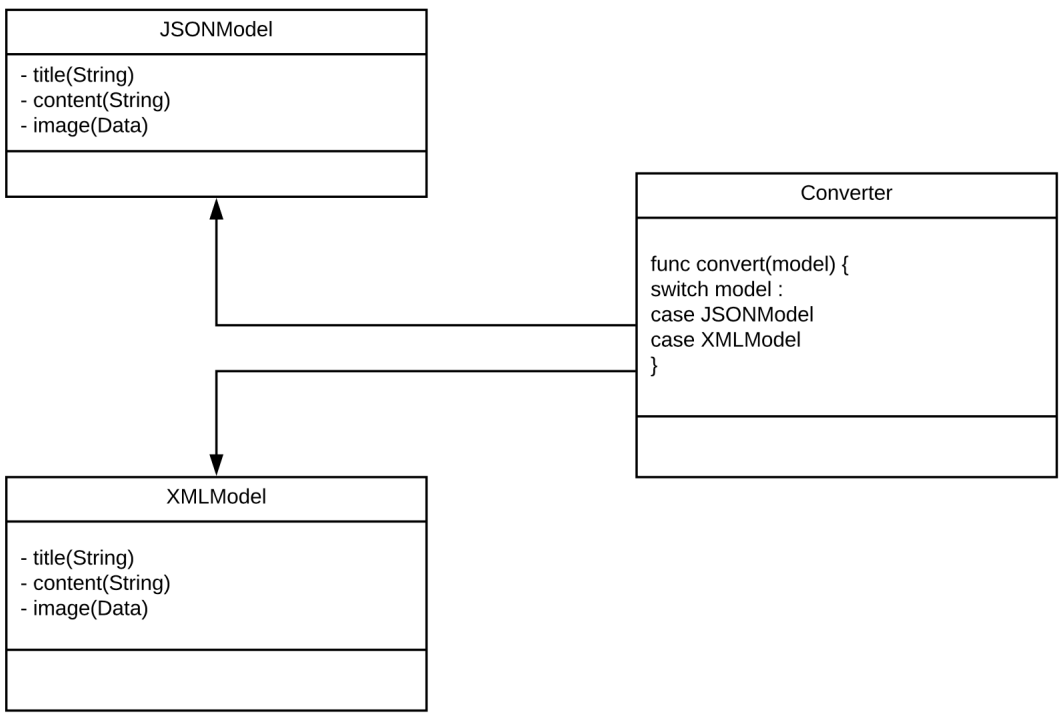
- **S** **S**ingle responsability.
- **O** **O**pen/Close.
- **L** **L**iskov substitution.
- **I** **I**nterface segregation.
- **D** **D**ependency inversion.

O Open/Close

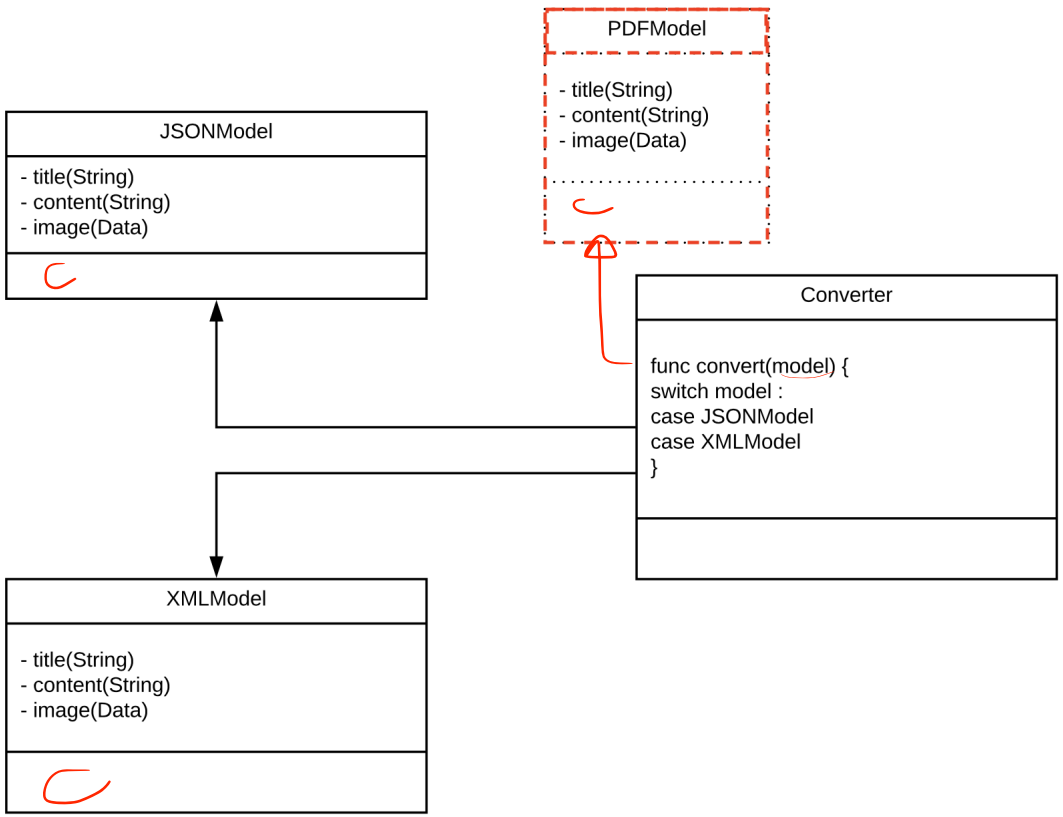
"Software entities should be open for extension and closed for modification"

- La modification d'une classe existante n'est pas permise.
- La modification d'une classe passe par l'extension de celle ci et la surcharge des méthodes concernées par le changement.
- Introduire une modification ajoute un nouveau niveau d'abstraction, ce qui augmente la complexité.

O Open/Close


JSONModel
- title(String)
- content(String)
- image(Data)
Converter
func convert(model) {
switch model :
case JSONModel
case XMLModel
}
XMLModel
- title(String)
- content(String)
- image(Data)

O Open/Close


PDFModel
- title(String)
- content(String)
- image(Data)
JSONModel
- title(String)
- content(String)
- image(Data)
Converter
func convert(model) {
switch model :
case JSONModel
case XMLModel
}
XMLModel
- title(String)
- content(String)
- image(Data)

O Open/Close

Il faut éviter :

- Le cast.
- Le switch.
- Le if/else.

O Open/Close

Il faut éviter :

- Le cast.
- Le switch.
- Le if/else.

La solution est d'ajouter une abstraction via une interface ou classe abstraite.

O Open/Close

Le principe OCP permet :

- d'ajouter des fonctionalitées sans modifier le code existant.
- de rendre la conception flexible et extensible.

Le principe SOLID

- **S** **S**ingle responsability.
- **O** **O**pen/Close.
- **L** **L**iskov substitution.
- **I** **I**nterface segregation.
- **D** **D**ependency inversion.

L Liskov substitution

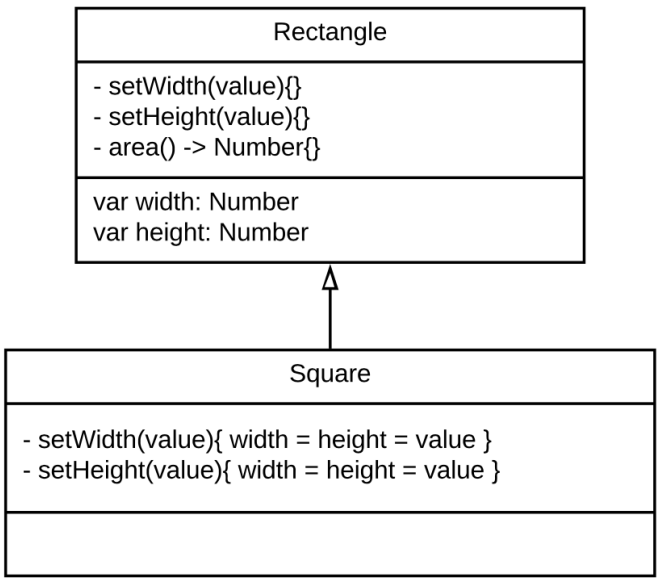
"Derived types must be completely substitutable for their base types".

Tout enfant ne doit jamais casser la définition de son parent.

- C'est une extension du principe de l'OCP.
- S'assurer que la classe dérivée étend la classe de base sans modifier son comportement.

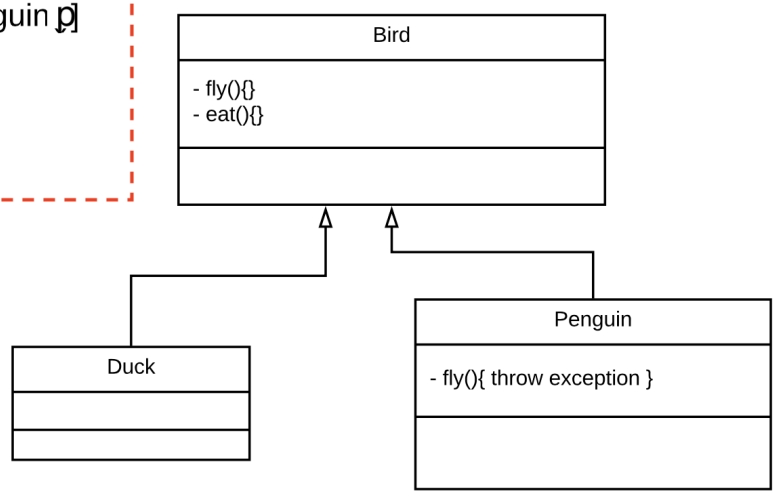
L Liskov substitution

```
TestArea(rect: Rectangle) {
  rect.setWidth(10)
  rect.setHeight(5)
  assert(rect.area == 50)
}
```

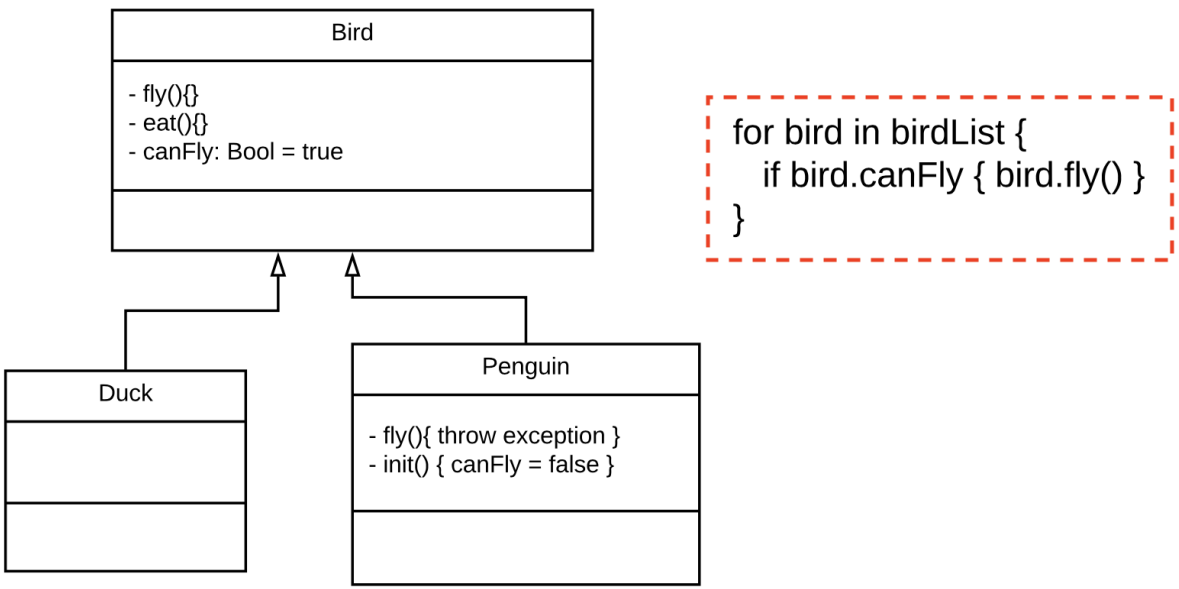


L Liskov substitution

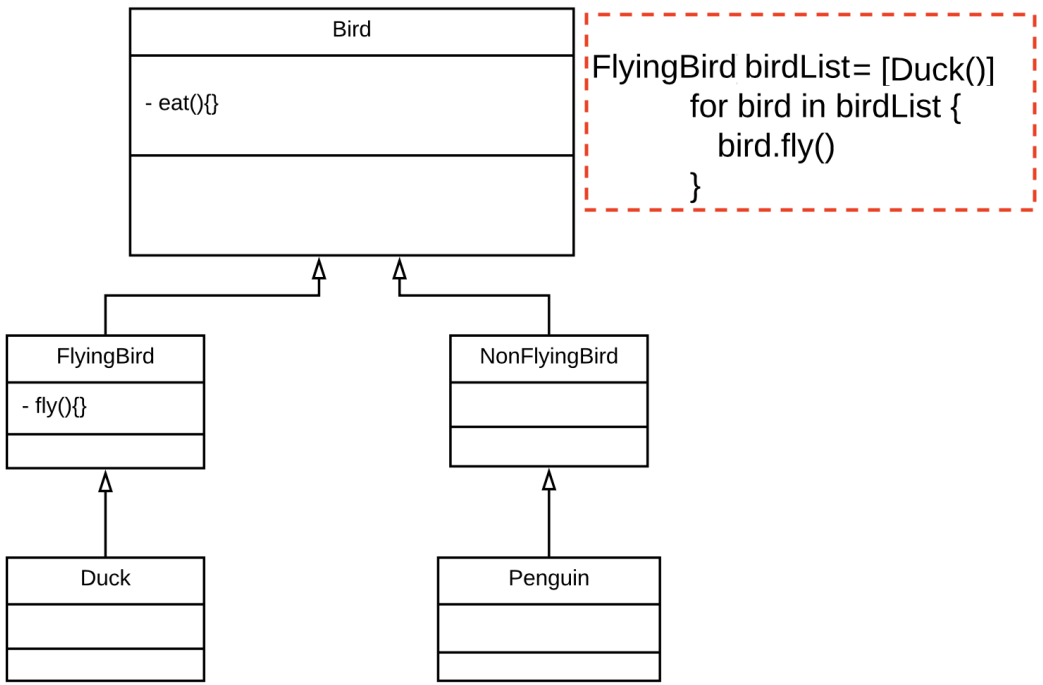
```
Bird birdList = [Duck d , Penguin p]

for bird in birdList {
   bird.fly()
}
```



L Liskov substitution



L Liskov substitution



L Liskov substitution

Le principe LSP permet :

- Bien comprendre la logique métier.
- Renforce le principe de l'héritage en POO qui se base sur les fonctionalités et non la définition traditionnelle de l'objet.

Le principe SOLID

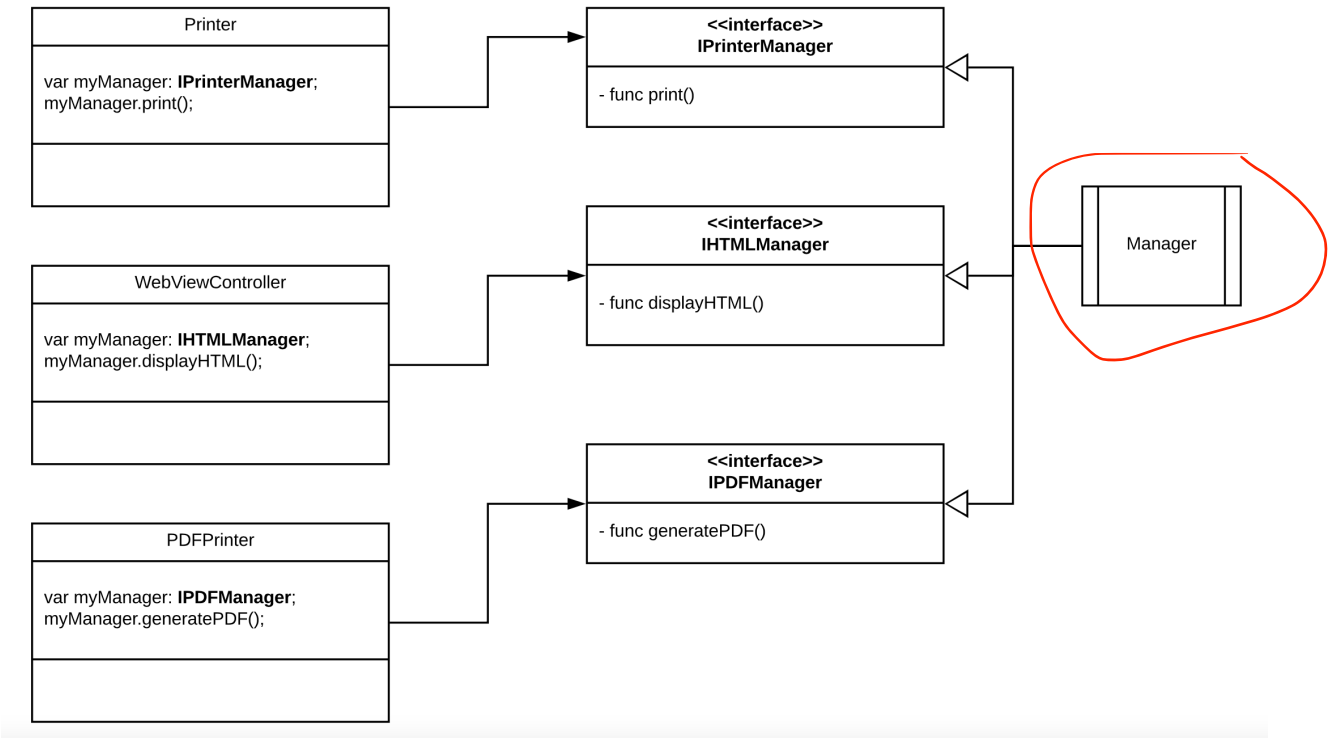
- **S** **S**ingle responsability.
- **O** **O**pen/Close.
- **L** **L**iskov substitution.
- **I** **I**nterface segregation.
- **D** **D**ependency inversion.

I Interface segregation

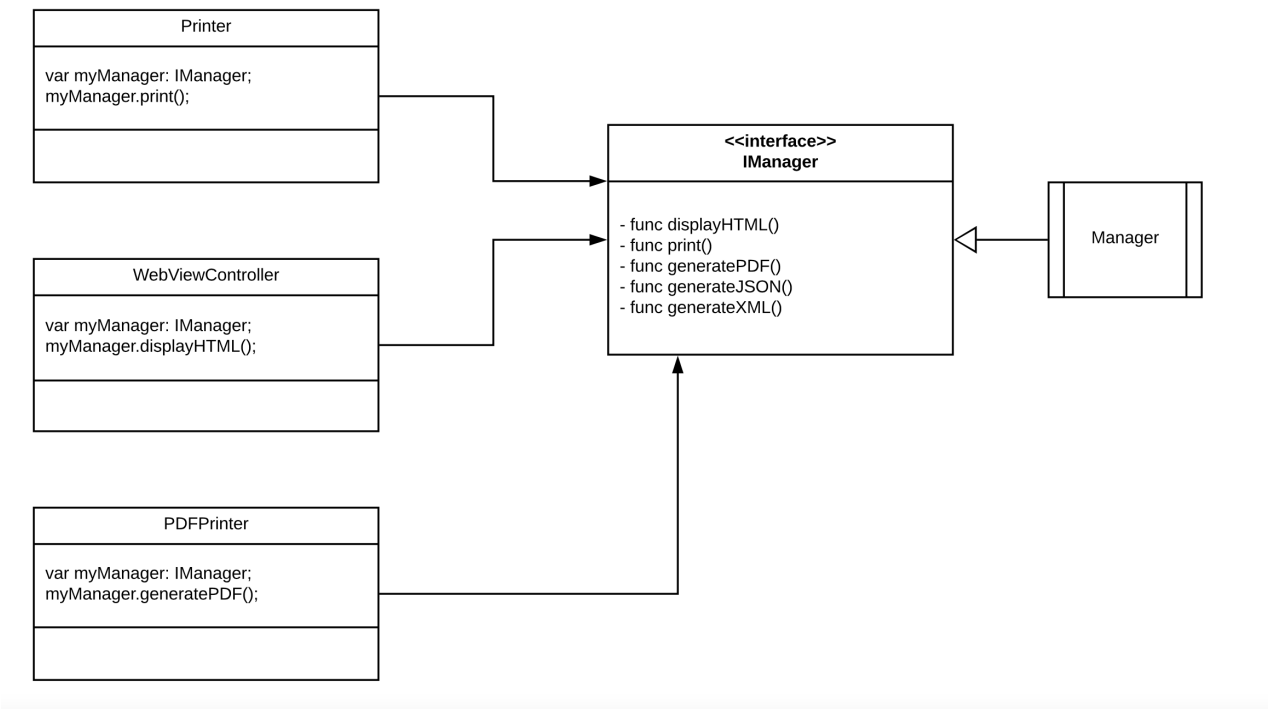
"Clients should not be forced to depend upon interfaces that they do not use".

- Une interface appartient a son client et non a son implémentation.
- Une classe ou module ne dépend que de l'interface qui l'intéresse.
-

I Interface segregation



I Interface segregation



I Interface segregation

Le principe ISP permet d'avoir :

- une conception faiblement couplée.
- une conception flexible et extensible.

Le principe SOLID

- **S** **S**ingle responsability.
- **O** **O**pen/Close.
- **L** **L**iskov substitution.
- **I** **I**nterface segregation.
- **D** **D**ependency inversion.

D Dependency inversion

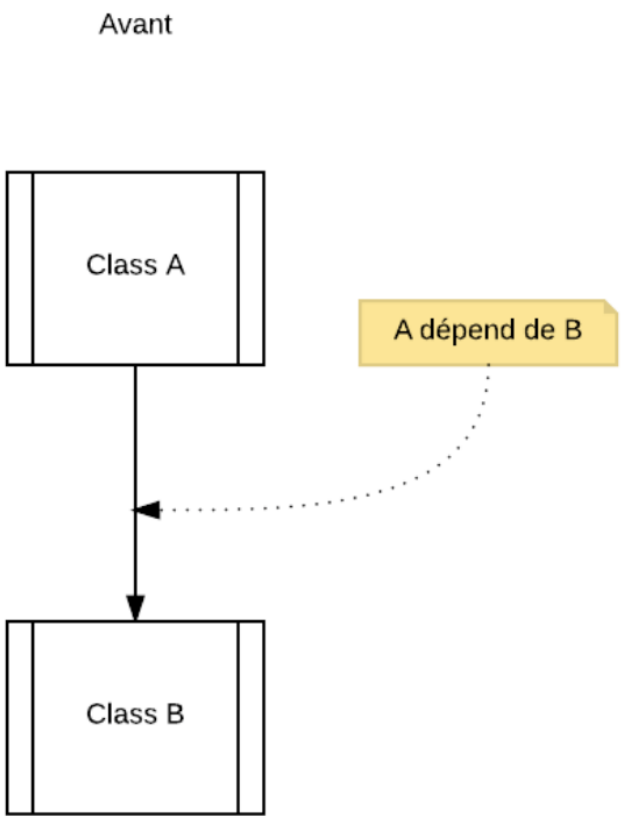
"Depend on abstractions, not on concretions".

- Les modules doivent dépendre entre eux par l'abstraction non par l'implémentation.
- Renverser le sens d'une dépendance.

D Dependency inversion

La dépendance :

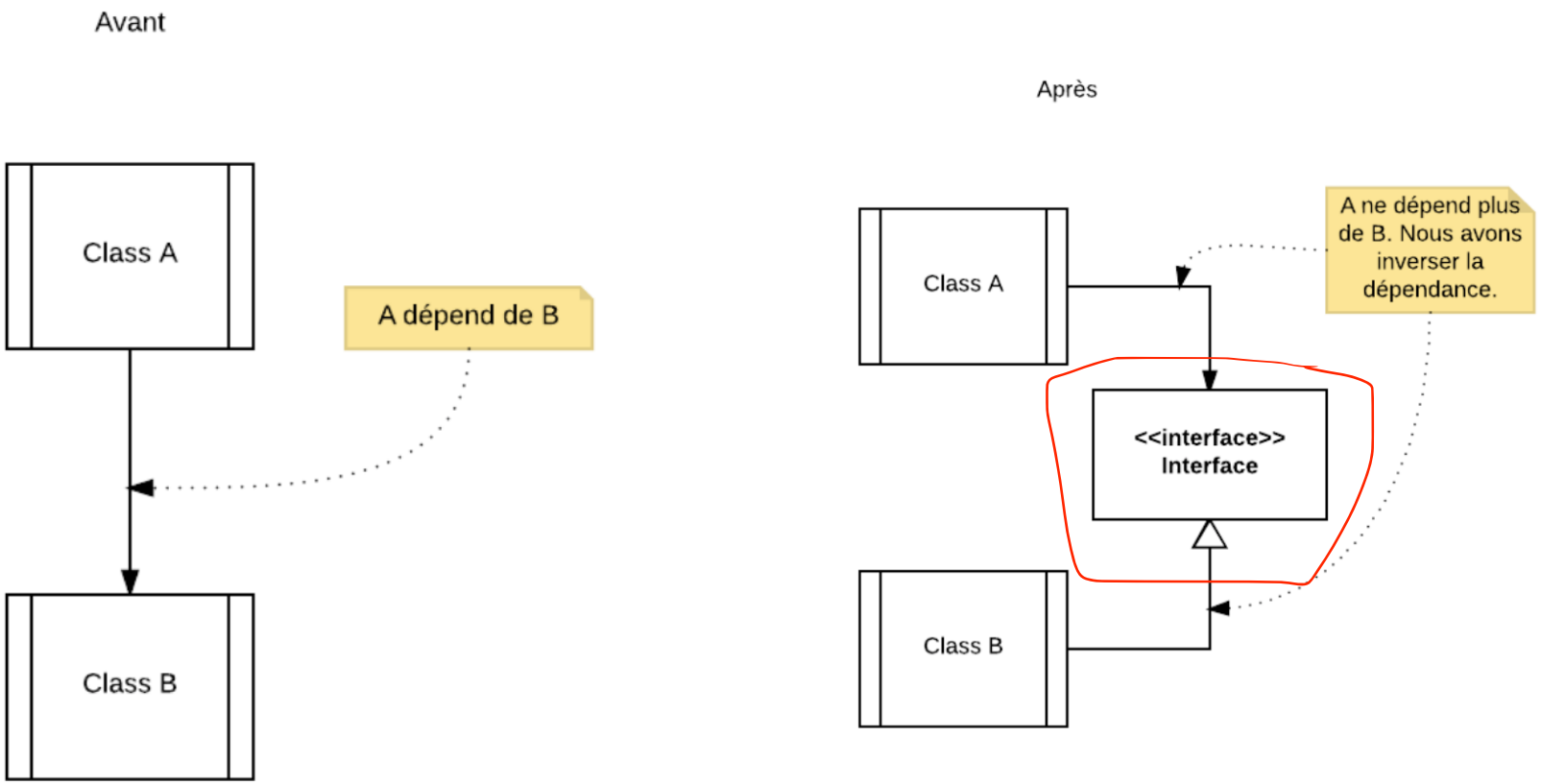
- A utilise un objet de B.
- A utilise une méthode de B.
- A utilise une variable de B.



Rendre A indépendant de B.

D Dependency inversion

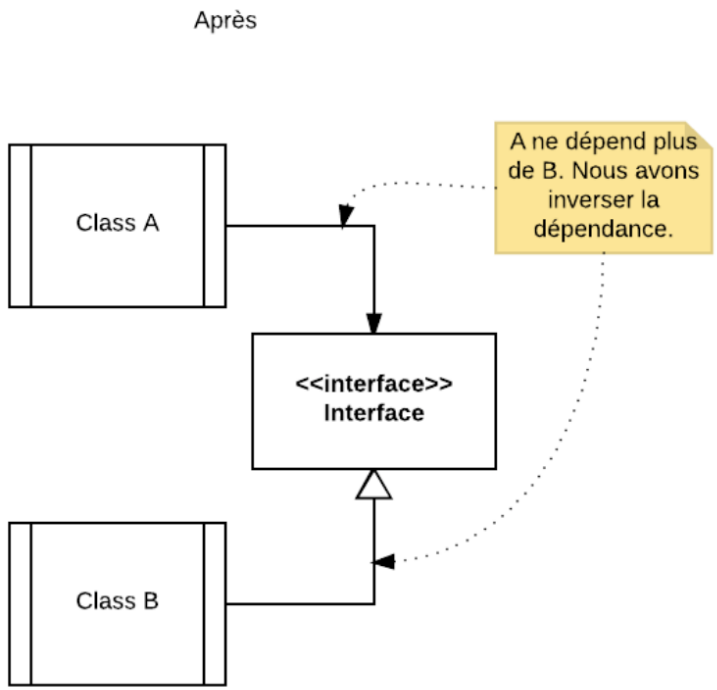
La dépendance inversée :



D Dependency inversion

La dépendance inversée :

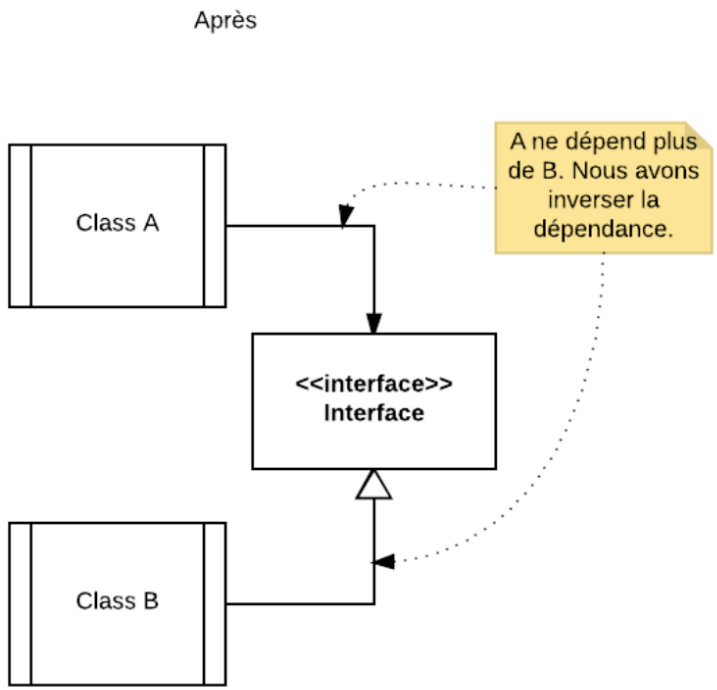
- Créer une interface I que B va implémenter.
- Changer toutes les références à B dans A par I.



D Dependency inversion

Les avantages :

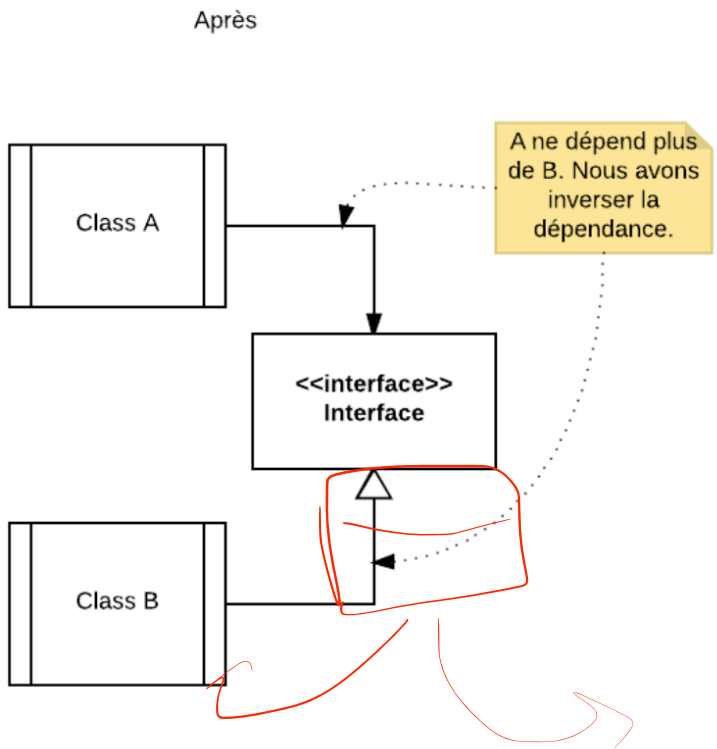
- A ne voit plus B.
- L'implémentation de B peut changer ou remplacer sans toucher A.
- Possibilité de tester A en utilisant le principe des Mocks.



D Dependency inversion

Les inconvénients :

- Trop d'abstraction rend une partie de l'application inutile.
-



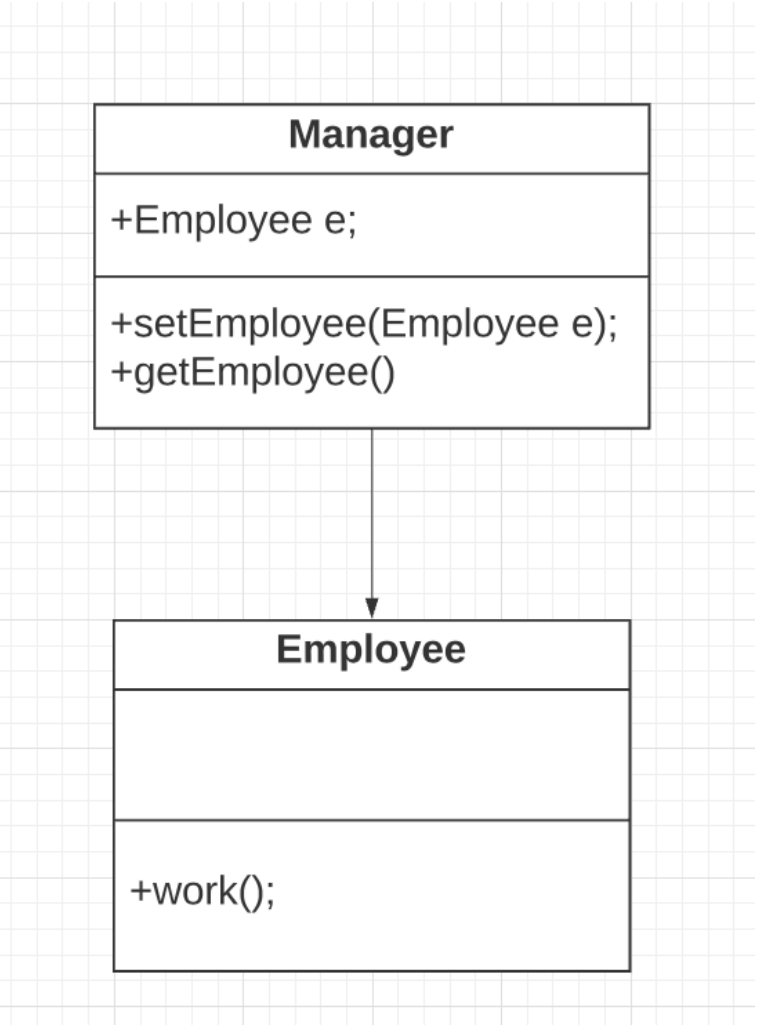
D Dependency inversion

Le guide de la création d'interfaces :

- les modules de haut niveaux ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- les abstractions ne doivent pas dépendre de détails mais l'inverse : les détails doivent dépendre d'abstractions.
- prendre en considération les modules et le sens des flèches qui franchissent les frontières entres modules.

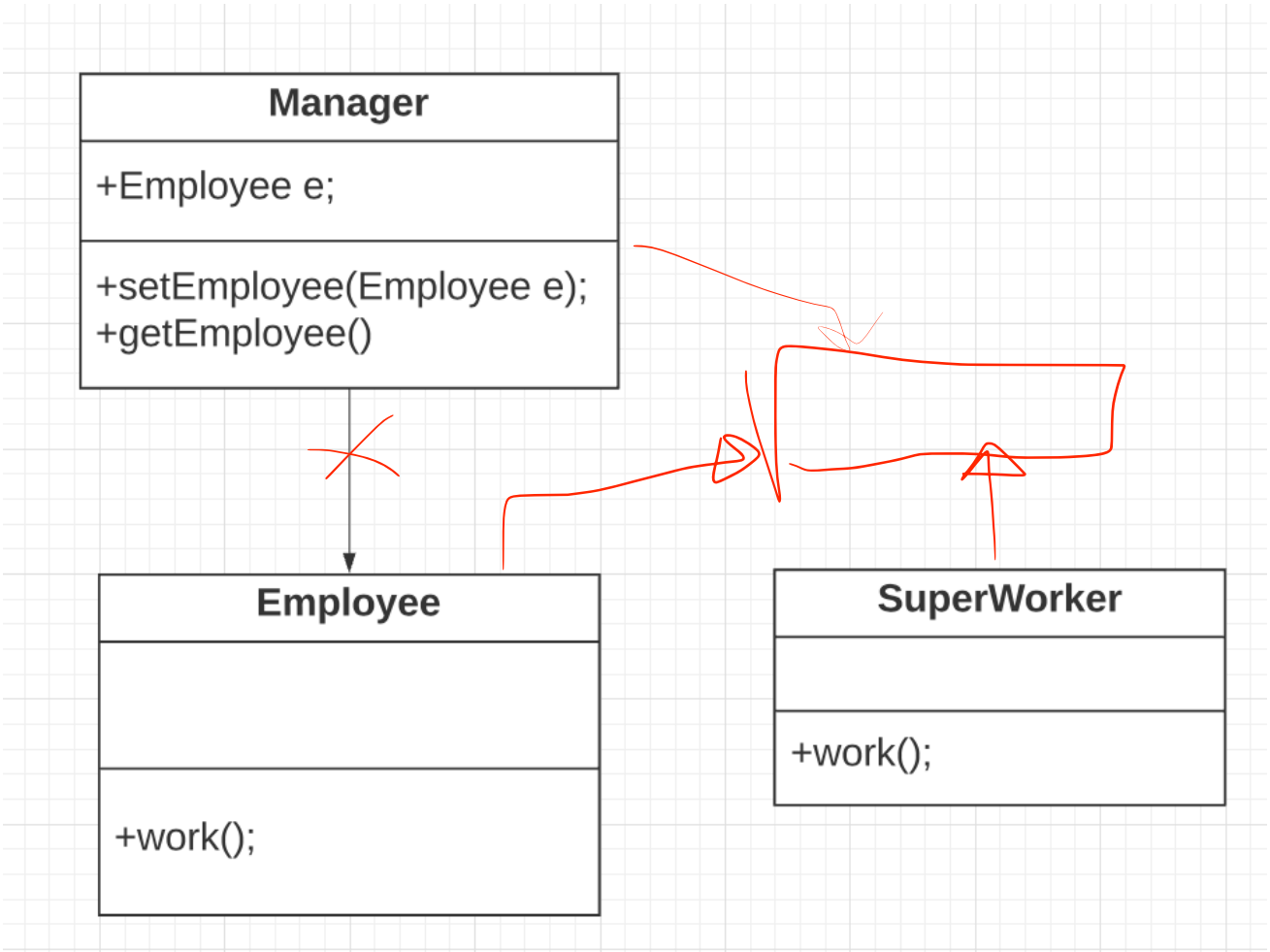
D Dependency inversion

Exemple :



D Dependency inversion

Exemple :



D Dependency inversion

Exemple :

```java
public class Employee {
  public void work() {
    // Do work
  }
}
```

```java
public class Manager {
  Employee employee;

  public void setEmployee(Employee employee
    this.employee = employee;
  }

  public void getEmployee() {
    employee.work();
  }
}
```

D Dependency inversion

Exemple :

```
public class Employee {
  public void work() {
    // Do work
  }
}
```

```
public class SuperWorke
  public void work() {
    // Do work
  }
}
```

```
public class Manager {
  Employee employee;

  public void setEmployee(Employee employee
    this.employee = employee;
  }

  public void getEmployee() {
    employee.work();
  }
}
```

D Dependency inversion

Exemple :

```java
public interface IEmployee {
  public void work();
}


public class BasicWorker implements IEmployee {
  @Override
  public void work() {
    // Do work
  }
}

public class BasicWorker implements IEmployee {
  @Override
  public void work() {
    // Do work
  }
}
```

Super

D Dependency inversion

Exemple :

```
public class Manager {
  IEmployee employee;

  public void setEmployee(IEmployee employee) {
    this.employee = employee;
  }

  public void getEmployee() {
    employee.work();
  }
}
```

Conclusion

- Commencer par le principe SRP pour définir les acteurs.
- Mettre en place le refactoring moins couteux et plus facile a mettre en place.
- OCP accroit l'aptitude au changement de votre application en brisant les couplages.
- Le DIP permet d'éviter le couplage fort.
-