

# Principes SOLID

ICT 301

Mahamat Ali

19 décembre 2025

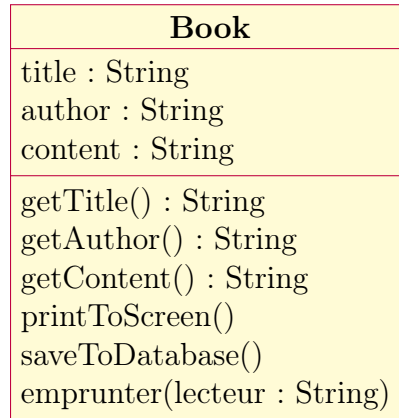
## Table des matières

<b>1</b>	<b>Single Responsibility Principle (SRP)</b>	<b>2</b>
1.1	Avant refactoring . . . . .	2
1.2	Après refactoring . . . . .	2
<b>2</b>	<b>Open Closed Principle (OCP)</b>	<b>2</b>
2.1	Avant refactoring . . . . .	2
2.2	Après refactoring . . . . .	3
<b>3</b>	<b>Liskov Substitution Principle (LSP)</b>	<b>3</b>
3.1	Avant refactoring . . . . .	3
3.2	Après refactoring . . . . .	3
<b>4</b>	<b>Interface Segregation Principle (ISP)</b>	<b>4</b>
4.1	Avant refactoring . . . . .	4
4.2	Après refactoring . . . . .	4
<b>5</b>	<b>Dependency Inversion Principle (DIP)</b>	<b>5</b>
5.1	Avant refactoring . . . . .	5
5.2	Après refactoring . . . . .	5

# 1 Single Responsibility Principle (SRP)

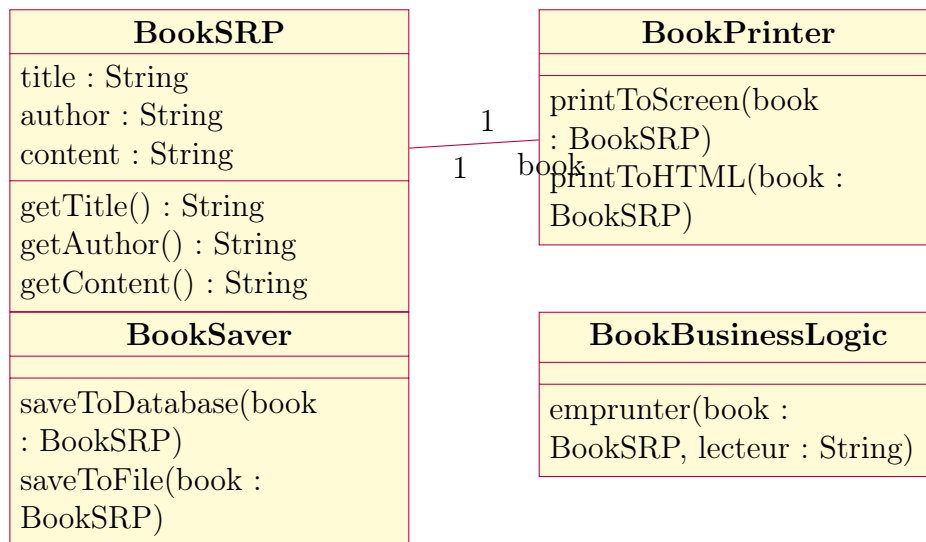
## 1.1 Avant refactoring

Le principe de responsabilité unique (SRP) stipule qu'une classe ne doit avoir qu'une seule raison de changer. Avant le refactoring, la classe suivante viole ce principe car elle gère plusieurs responsabilités à la fois.



## 1.2 Après refactoring

Après refactoring, chaque responsabilité est séparée dans une classe distincte, ce qui permet de respecter pleinement le principe SRP.



# 2 Open Closed Principle (OCP)

## 2.1 Avant refactoring

Le principe Open/Closed stipule qu'une classe doit être ouverte à l'extension mais fermée à la modification. Avant le refactoring, la classe suivante viole ce principe car chaque nouveau type de client nécessite une modification du code.

DiscountCalculator_Avant
calculateDiscount(customerType : String, amount : double)

## 2.2 Après refactoring

Après refactoring, le comportement est rendu extensible grâce au polymorphisme. L'ajout d'un nouveau type de réduction se fait sans modifier les classes existantes.

DiscountStrategy
calculate(amount : double)

DiscountCalculator
calculateDiscount(amount : double)

StudentDiscount	VipDiscount	RegularDiscount
calculate(amount : double)	calculate(amount : double)	calculate(amount : double)

## 3 Liskov Substitution Principle (LSP)

### 3.1 Avant refactoring

Le principe de substitution de Liskov stipule qu'une classe dérivée doit pouvoir être utilisée à la place de sa classe de base sans altérer le comportement du programme. Dans cet exemple, la classe *Square* hérite de *Rectangle* et modifie son comportement attendu, ce qui viole le principe LSP.

Rectangle_Avant
width : int height : int
setWidth(width : int) setHeight(height : int) getArea() : int

Square
setWidth(width : int) setHeight(height : int)

### 3.2 Après refactoring

Après refactoring, les formes géométriques implémentent une abstraction commune sans relation d'héritage incorrecte, ce qui garantit le respect du principe LSP.

Shape
getArea() : int

Rectangle
width : int height : int
getArea() : int

Square
side : int
getArea() : int

## 4 Interface Segregation Principle (ISP)

### 4.1 Avant refactoring

Le principe de ségrégation des interfaces stipule qu'une classe ne doit pas être forcée d'implémenter des méthodes qu'elle n'utilise pas. Dans cet exemple, une interface trop large oblige certaines classes à implémenter des méthodes inutiles.

Machine_Avant	SimplePrinter
print() scan() fax()	print() scan() fax()

### 4.2 Après refactoring

Après refactoring, les interfaces sont séparées selon leurs responsabilités, permettant à chaque classe d'implémenter uniquement les fonctionnalités nécessaires.

Printer	Scanner	Fax
print()	scan()	fax()

SimplePrinter	MultiFunctionPrinter
print()	print() scan() fax()

## 5 Dependency Inversion Principle (DIP)

### 5.1 Avant refactoring

Le principe d'inversion des dépendances stipule que les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais d'abstractions. Dans cet exemple, le module de haut niveau dépend directement d'une classe concrète.

### 5.2 Après refactoring

Après refactoring, le module de haut niveau dépend d'une abstraction, ce qui permet de respecter le principe DIP.

Switchable
turnOn() turnOff()

Switch
device : Switchable operate()

Light
turnOn() turnOff()