Université de Monastir
Institut Supérieur d'Informatique et de Mathématiques
Département Informatique

# Cours
# Design Patterns et conception par contrats

ING 2 - Génie Logiciel

# Responsable du cours

Nom & Prénom : Dr. Sameh HBAIEB
Grade: Maître Assistant
Département : Informatique
Université : ISIMM-Université de Monastir


Email : samehhbaieb11@gmail.com

LinkedIn: https://www.linkedin.com/in/sameh-hbaieb-b6aab61b9/

Facebook: https://www.facebook.com/sameh.hbaieb06

Bureau : [A38, Bloc A]

Disponibilités :
- Permanence pédagogique : Mardi et jeudi
- Rendez-vous par email

# Plan du cours

01 Principes de conception Orientée Objets

02 Introduction aux Design Patterns

03 Patterns de Création

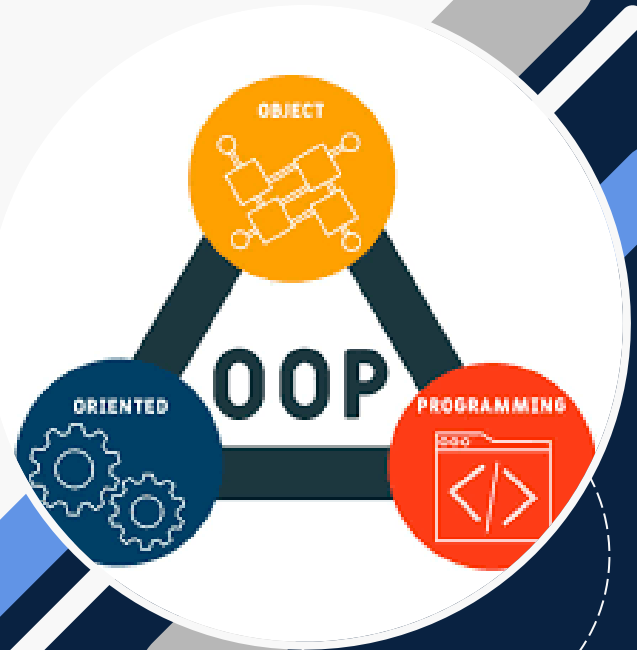
04 Patterns Structurels

05 Patterns Comportementaux

“Ce cours est conçu pour vous aider à acquérir une maîtrise pratique des **Design Patterns** et des **principes SOLID**. N’hésitez pas à poser des questions et à demander des exemples supplémentaires”

01

# Principes de conception Orientée Objets

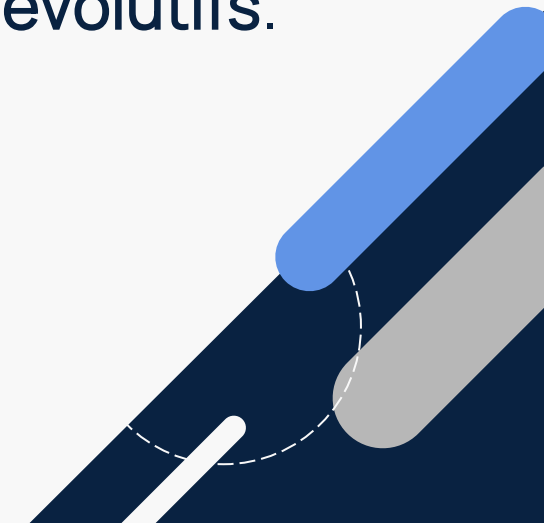


# Objectifs de ce chapitre

- Réviser les fondamentaux de la conception orientée objet.
- Comprendre les notions de **couplage**, **cohésion**, **abstraction**, **responsabilité unique**.
- Introduire les principes **SOLID** qui guideront l'utilisation des Design Patterns.

# Qu’est ce que le génie logiciel ?

- Le génie logiciel est l’ensemble des méthodes, outils et bonnes pratiques permettant de concevoir, développer, tester, déployer et maintenir des systèmes logiciels **complexes**, **fiables** et **évolutifs**.



# Les grands principes du génie logiciel

## 1. Modularité

Principe : Diviser le logiciel en modules indépendants avec une responsabilité unique.

Exemple :

Dans une application e-commerce :

Module Paiement → gère uniquement les transactions.

Module Catalogue → gère uniquement la liste des produits.

Module Utilisateur → gère l'inscription, l'authentification, le profil.

. . . . .

**Avantage :** Chaque module peut être développé et testé indépendamment.

# Les grands principes du génie logiciel

## 2. Abstraction

Principe : Masquer les détails d'implémentation et ne montrer que l'essentiel.

Exemple :

```
interface PaymentMethod {
   void pay(double amount);
}
class CreditCardPayment implements PaymentMethod { ... }
class PaypalPayment implements PaymentMethod { ... }
```

**Avantage :** L'utilisateur du module Payment n'a pas besoin de connaître les détails de CreditCardPayment

# Les grands principes du génie logiciel

## 3. Cohésion et Couplage

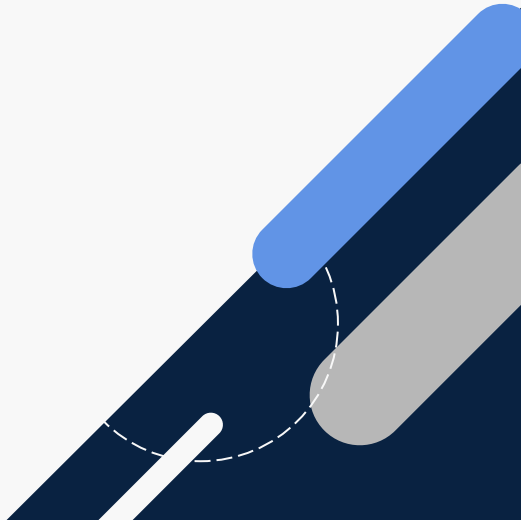
Cohésion : Chaque module/fonction a une seule responsabilité

Couplage : Minimiser les dépendances entre modules.

Exemple :

Mauvais design :

```
class UserManager {
   void saveUser() { ... }
   void sendEmail() { ... }
   void logActivity() { ... }
}
```

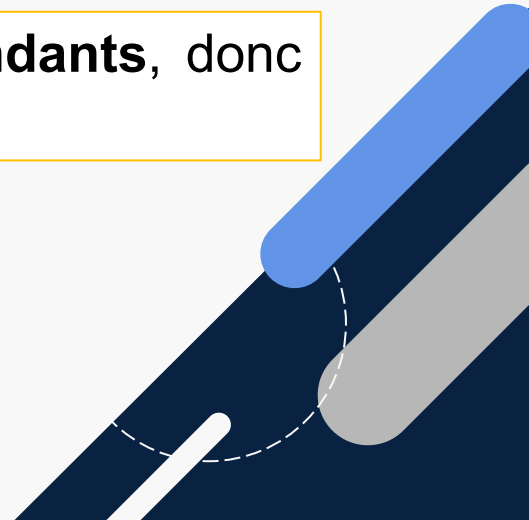


# Les grands principes du génie logiciel

Bon design :

```
class UserManager { void saveUser() { ... } }
class EmailService { void sendEmail() { ... } }
class Logger { void logActivity() { ... } }
```

**Avantage :** Les modules sont **cohérents et indépendants**, donc faciles à maintenir.



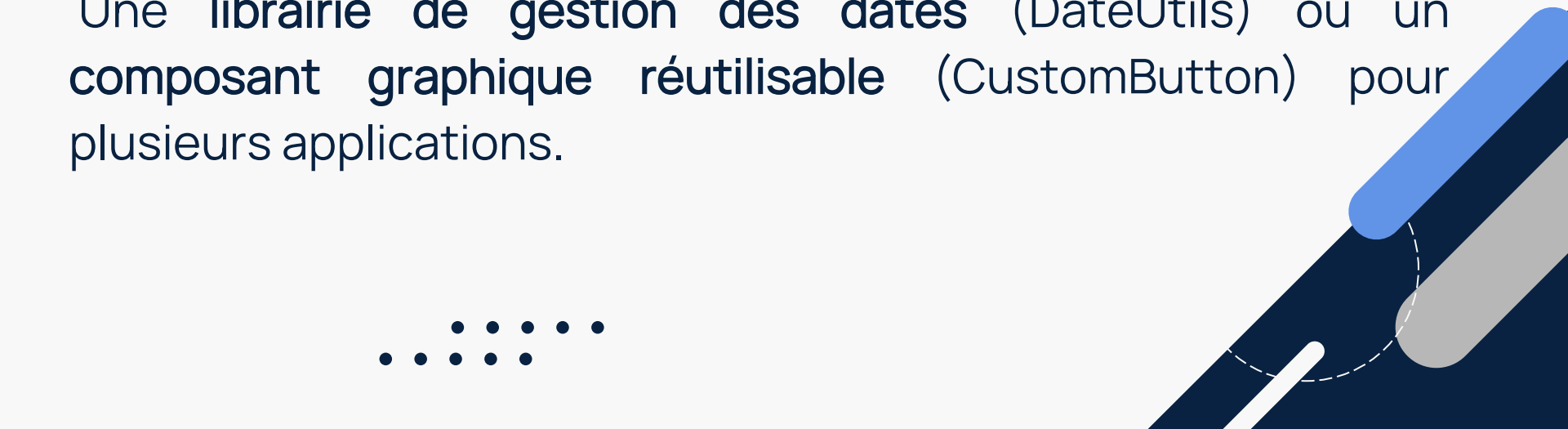
# Les grands principes du génie logiciel

## 4. Réutilisabilité

Principe : Créer des composants qui peuvent être utilisés dans plusieurs projets.

Exemple :

Une **librairie de gestion des dates** (DateUtils) ou un **composant graphique réutilisable** (CustomButton) pour plusieurs applications.



# Les grands principes du génie logiciel

## 5. Portabilité

Principe : Le logiciel doit fonctionner sur plusieurs plateformes.

Exemple :

Application Java : fonctionne sur Windows, Linux et MacOS grâce à la JVM.

Application web : compatible avec Chrome, Firefox et Edge.

# Les grands principes du génie logiciel

## 6. Extensibilité

Principe : Ajouter de nouvelles fonctionnalités sans modifier le code existant.

Exemple :

Ajouter un nouveau moyen de paiement à une application e-commerce.

# Les grands principes du génie logiciel

## 7. Maintenabilité

Principe : Faciliter la correction, l'évolution et l'adaptation du logiciel.

Exemple :

Un code **clair et commenté**, avec tests unitaires (JUnit) et suivi de version (Git) permet de corriger un bug rapidement.

# Les grands principes du génie logiciel

## 8. Fiabilité

Principe : Logiciel qui fonctionne correctement même en cas d'erreurs ou d'usage inattendu.

Exemple :

Une application bancaire : vérifie les soldes avant de transférer l'argent et gère les exceptions réseau.

Utilisation de **try/catch**, **tests automatisés** et **contrôle des entrées**.

# Les grands principes du génie logiciel

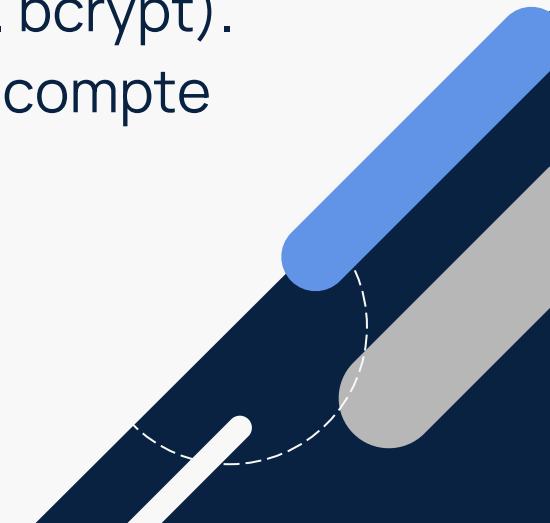
## 9. Sécurité

Principe : Protéger le logiciel contre les accès non autorisés et les fuites de données.

Exemple :

Stockage des mots de passe avec hachage (ex. bcrypt).

Authentification à deux facteurs (2FA) pour un compte utilisateur.



# Les grands principes du génie logiciel

## 10. Économie et Efficacité

Principe : Développer un logiciel en optimisant les coûts et les ressources.

Exemple :

Utilisation de **méthodes agiles** pour livrer rapidement un MVP.

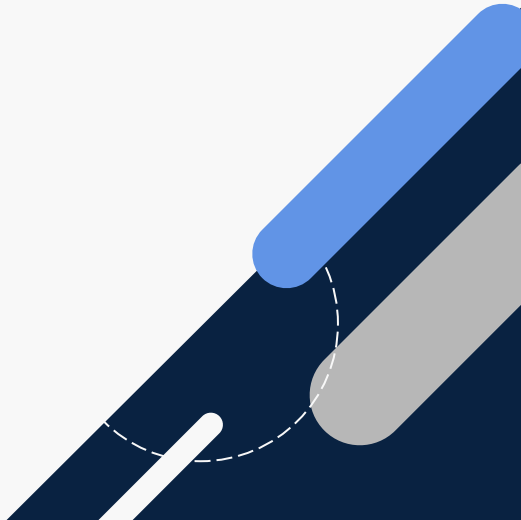
Intégration continue (CI/CD) pour éviter les retards et réduire les coûts de correction.

# Les grands principes du génie logiciel

- En conclusion:

Le **génie logiciel** fournit des **principes généraux** :

- Cohésion forte, couplage faible
- Modularité, réutilisabilité
- Abstraction et encapsulation
- Séparation des préoccupations

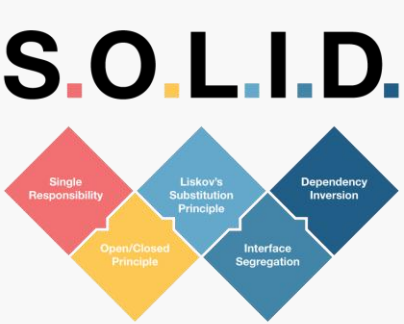


“Chaque principe que vous découvrez est une arme contre la **complexité logicielle**.”

“L’objectif n’est pas seulement de coder… mais de coder **intelligemment** et **durablement**..”

# Des grands principes aux principes SOLID

👉 **Comment appliquer concrètement** les principes OO dans le développement logiciel ?




**SOLID** = 5 principes proposés par Robert C. Martin

**S** → Single Responsibility Principle (SRP)

**O** → Open/Closed Principle (OCP)

**L** → Liskov Substitution Principle (LSP)

**I** → Interface Segregation Principle (ISP)

**D** → Dependency Inversion Principle (DIP)

# Principe 1. Responsabilité unique

**Définition** : Une classe doit avoir une seule raison de changer.

**But** : améliorer la cohésion, faciliter la maintenance.

**Exemple** :

```
class Report {
  void calculateStatistics() {
    // Calcul des stats
  }
  void printReport() {
    // Impression du rapport
  }
void saveToFile() {
    // Sauvegarde dans un fichier
  }
}
```

# Principe 1. Responsabilité unique

Problèmes :

- **3 responsabilités différentes** : calcul, impression, persistance.
- Toute modification (nouveau format d'impression, nouvelle base de données, changement de logique métier) implique de **modifier la même classe**.
- Forte probabilité d'erreurs et faible maintenabilité.

**= > Respect du SRP:**

On sépare les responsabilités en plusieurs classes cohésives :
StatisticsCalculator, ReportPrinter, ReportSaver

# Principe 2. Ouverture / Fermeture

**Définition :** Une classe doit être ouverte à l'extension, mais fermée à la modification.

**But :**

- On doit pouvoir **ajouter de nouvelles fonctionnalités** sans modifier le code existant.
- On favorise **l'héritage, le polymorphisme ou les interfaces** au lieu de modifier directement une classe.

# Principe 2. Ouverture / Fermeture

## Exemple: Violation OCP

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

   public Rectangle(double w, double h) {
      this.width = w;
      this.height = h;
   }
}
class Circle {
   double radius;

   public Circle(double r) {
      this.radius = r;
   } } }
```

```
class AreaCalculator {
   public double calculate(Object[] shapes) {
      double area = 0;
      for (Object shape : shapes) {
         if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            area += r.width * r.height;
         } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            area += Math.PI * c.radius * c.radius;
         }
         // Si j'ajoute Triangle → je dois modifier cette classe
      }
      return area;
   } }
```

# Principe 2. Ouverture / Fermeture

## Respect OCP

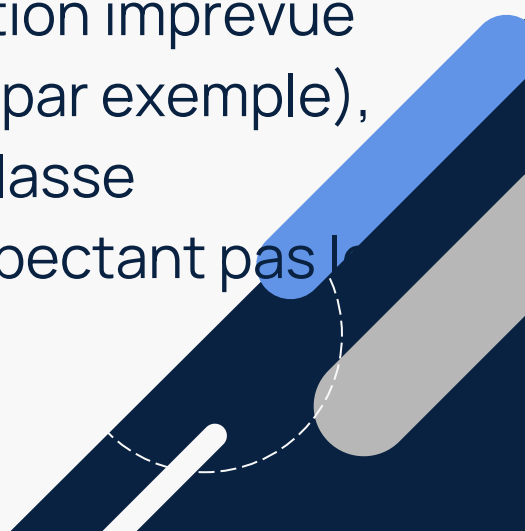
```
// Abstraction commune
interface Shape {
   double area();
}
// Rectangle
class Rectangle implements Shape {
   double width, height;
   public Rectangle(double w, double h) {
      this.width = w;
      this.height = h;
   }
   public double area() {
      return width * height;
   }
}
```

```
// Cercle
class Circle implements Shape {
   double radius;
   public Circle(double r) {
      this.radius = r;
   }
   public double area() {
      return Math.PI * radius * radius;
   } }
// Calculateur qui respecte OCP
class AreaCalculator {
   public double calculate(Shape[] shapes) {
      double area = 0;
      for (Shape shape : shapes) {
         area += shape.area();
      }
return area;    } }
```

# Principe 3. Substitution de Liskov

- Les sous classes doivent pouvoir être substituées à leur classe de base sans modifier la logique métier du programme (transparence vis-à-vis les utilisateurs)

- Cela signifie qu'il ne faut pas lever d'exception imprévue (comme UnsupportedOperationException par exemple), ou modifier les valeurs des attributs de la classe principale d'une manière inadaptée, ne respectant pas le contrat défini par la méthode.



# Principe 3. Substitution de Liskov

## Exemple: Violation de LSP

```
class Rectangle {
   protected int width;
   protected int height;

   public void setWidth(int w) { this.width = w; }
   public void setHeight(int h) { this.height = h; }

   public int getArea() { return width * height; }
}
```

```
class Square extends Rectangle {
   @Override
   public void setWidth(int w) {
      this.width = w;
      this.height = w;  // forcer un carré
   }

   @Override
   public void setHeight(int h) {
      this.width = h;
      this.height = h; // forcer un carré
   }
}
```

# Principe 3. Substitution de Liskov

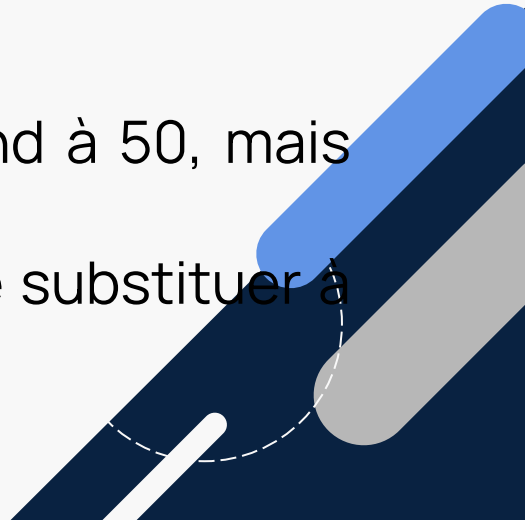
Problème :

- Square hérite de Rectangle, mais ne respecte pas son comportement attendu.

Exemple d'utilisation :

```
Rectangle r = new Square();
r.setWidth(5);
r.setHeight(10);
System.out.println(r.getArea()); // On s'attend à 50, mais on obtient 100
```

**LSP violé** : un Square ne peut pas toujours se substituer à un Rectangle.



# Principe 3. Substitution de Liskov

## Respect LSP:

```
interface Shape {
   int getArea();
}
// Rectangle indépendant
class Rectangle implements Shape {
   private int width;
   private int height;
   public Rectangle(int w, int h) {
      this.width = w;
      this.height = h;
   }
   public int getArea() {
      return width * height;
   } }
```

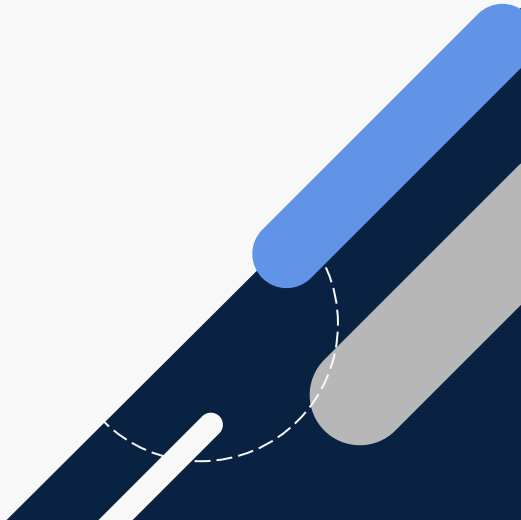
```
// Carré indépendant
class Square implements Shape {
   private int side;

   public Square(int s) {
      this.side = s;
   }

   public int getArea() {
      return side * side;
   }
}
```

# Principe 3. Substitution de Liskov

## Avantages :

- Chaque classe respecte son comportement attendu.
- Square et Rectangle sont **au même niveau** hiérarchique.
- On peut utiliser :

```
Shape[] shapes = { new Rectangle(5, 10), new Square(7) };
for (Shape s : shapes) {
   System.out.println(s.getArea());
}
// Résultats corrects pour chaque forme
```



# Principe 4. Ségrégation d'interface

**Définition** : Mieux vaut plusieurs interfaces spécifiques qu'une interface générale trop large.

- Le principe stipule que le client ne doit voir que les services dont il a besoin.
- Autrement dit, la dépendance d'une classe vers une autre doit être restreinte à l'interface la plus petite possible.

# Principe 4. Ségrégation d'interface

## Violation de l'ISP :

```
// Interface trop large
interface Worker {
    void work();
    void eat();
    void sleep();
}
```

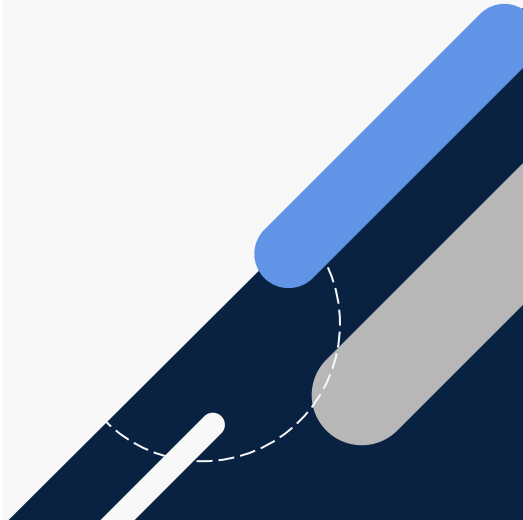
```
class HumanWorker implements Worker {
    public void work() { System.out.println("L'humain travaille..."); }
    public void eat() { System.out.println("L'humain mange..."); }
    public void sleep() { System.out.println("L'humain dort..."); }
}
```

```
class RobotWorker implements Worker {
    public void work() { System.out.println("Le robot travaille..."); }

    // Problème : un robot ne mange pas et ne dort pas
    public void eat() { throw new UnsupportedOperationException("Le
robot ne mange pas"); }
    public void sleep() { throw new UnsupportedOperationException("Le
robot ne dort pas"); }
}
```

# Principe 4. Ségrégation d'interface

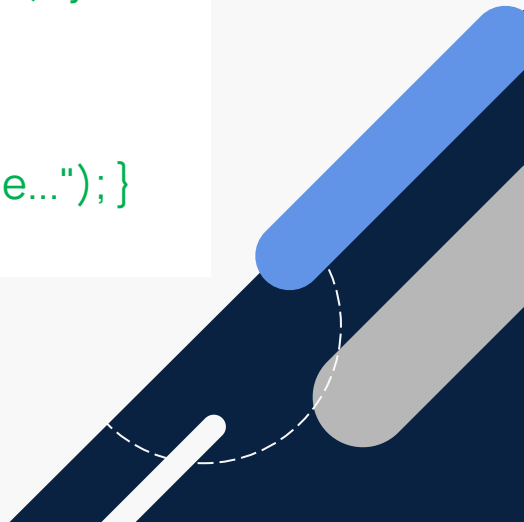
**Respect de l'ISP** : On **segmente l'interface** en plusieurs petites interfaces cohérentes :

```
// Interfaces spécialisées
interface Workable {
   void work();
}

interface Eatable {
   void eat();
}

interface Sleepable {
   void sleep();
}
```



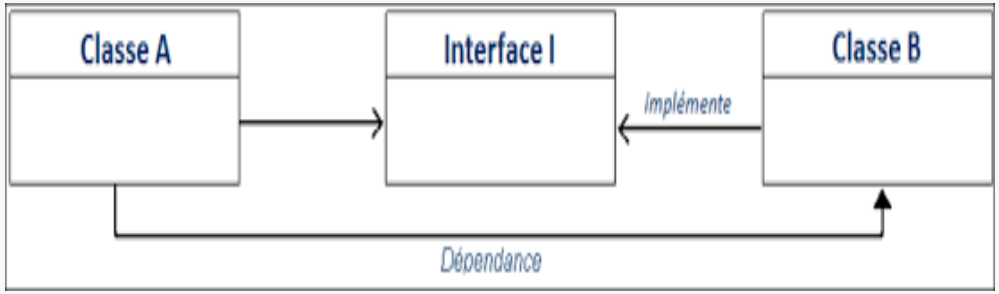
# Principe 4. Ségrégation d'interface

```
// Implémentations
class HumanWorker implements Workable, Eatable, Sleepable {
   public void work() { System.out.println("L'humain travaille..."); }
   public void eat() { System.out.println("L'humain mange..."); }
   public void sleep() { System.out.println("L'humain dort..."); }
}

class RobotWorker implements Workable {
   public void work() { System.out.println("Le robot travaille..."); }
}
```



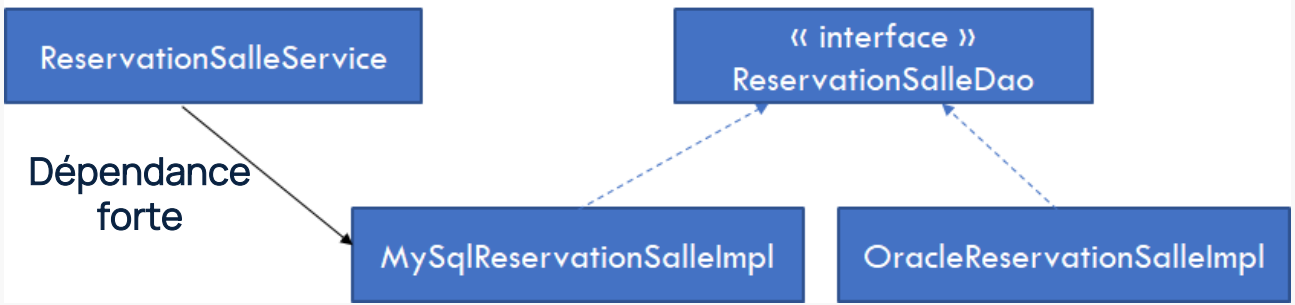
# Principe 5. Inversion des dépendances

- Dépendre des abstractions, pas des implémentations.
- Ce principe vise principalement à réduire les dépendances entre les modules de code (faible couplage).



# Principe 5. Inversion des dépendances

Exemple avec dépendance forte:




```
public class ReservationSalleService {
 private ReservationSalleDao reservationSalleDao;

 public ReservationSalleService() {
  reservationSalleDao = new MySqlReservationSalleImpl();
 }
 public void reserver(ReservationSalle reservationSalle) {
  // faire un traitement nécessaire
  // (par exemple la validation de la réservation)
  // sauvegarder la réservation
  reservationSalleDao.save(reservationSalle);
 } }
```

ReservationSalleService dépend de ReservationSalleDao

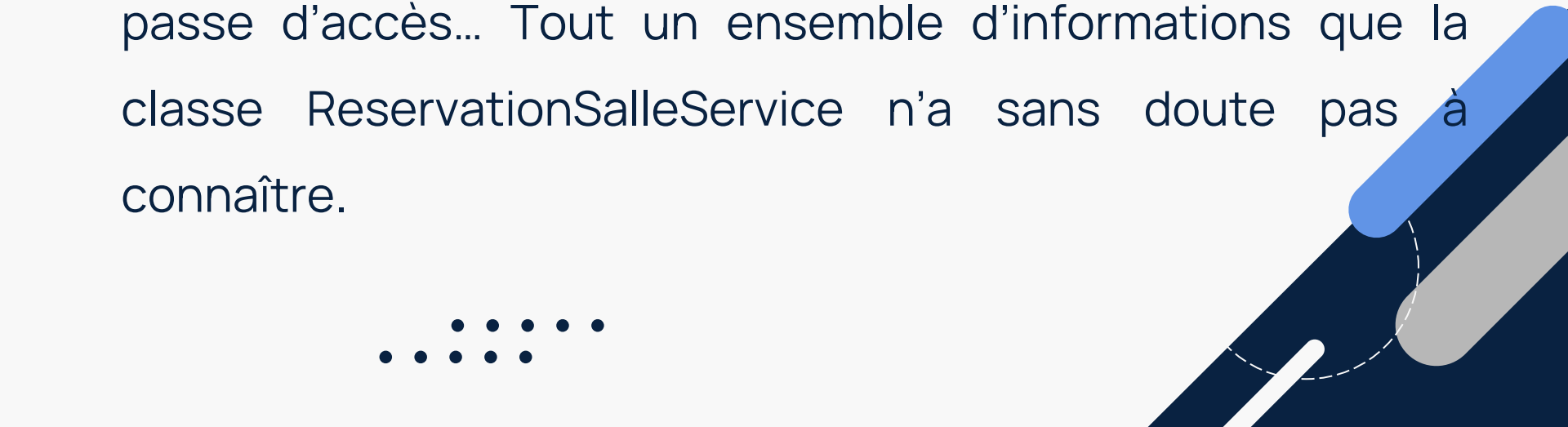
Création d'une instance de ReservationSalleDao dans le constructeur. La classe doit connaître le type d'implémentation (objet concrêt)
A éviter l'instanciation de l'objet reservationSalleDao

# Principe 5. Inversion des dépendances

- Cette solution est problématique.
- Il existe un **couplage fort** entre les objets: ReservationSalleService et ReservationSalleDao:
- il n'est pas possible de considérer ReservationSalleDao comme une abstraction ou une interface puisque le service doit lui-même spécifier le type concret de l'attribut MySqlReservationSalleImpl.

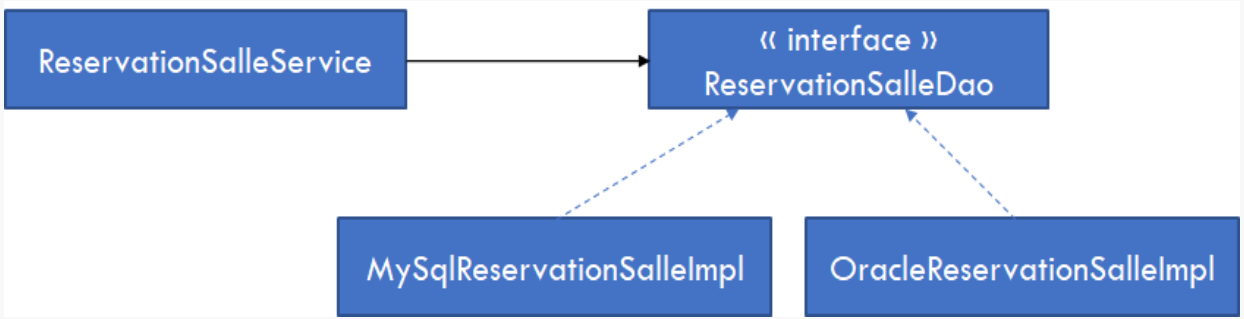
# Principe 5. Inversion des dépendances

- La création d'un objet de type MySqlReservationSalleImpl nécessite de passer des paramètres concrets comme par exemple l'adresse de la base de données, le login, le mot de passe d'accès… Tout un ensemble d'informations que la classe ReservationSalleService n'a sans doute pas à connaître.



# Principe 5. Inversion des dépendances

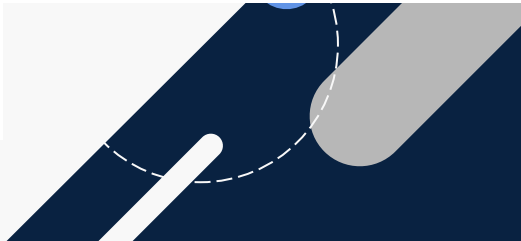
**Solution**: Inverser les dépendances pour réduire le couplage entre les classes



# Principe 5. Inversion des dépendances

```
public class ReservationSalleService {

 private ReservationSalleDao reservationSalleDao;

 public ReservationSalleService(ReservationSalleDao reservationSalleDao) {
  this.reservationSalleDao = reservationSalleDao;
 }

 public void reserver(ReservationSalle reservationSalle) {
  // faire un traitement nécessaire
  // (par exemple la validation de la réservation)

  // sauvegarder la réservation
  reservationSalleDao.save(reservationSalle);
 } }
```

Injection de la dépendance par le constructeur:
ReservationSalleService ne dépend plus d'un objet concret ReservationSalleDao.
On fait passer comme paramètre un objet abstrait qu'on ne connaît pas son type



# Les principes SOLID

| Principe | Idée clé | Objectif |
|---|---|---|
| SRP | Une seule responsabilité | Cohésion |
| OCP | Ouvert à l'extension, fermé à la modification | Extensibilité |
| LSP | Remplaçabilité des sous-classes | Polymorphisme correct |
| ISP | Interfaces spécifiques | Faible couplage |
| DIP | Dépendances sur abstractions | Flexibilité |