## **Résumé – Application des principes SOLID**

### **1️ SRP – Single Responsibility Principle**

**Avant :**

* Classe CarManager avec plusieurs responsabilités :  
  + Accès aux données (getFromDb)
  + Formatage (getCarsNames)
  + Logique métier (getBestCar)
* Fort couplage, code difficile à maintenir et à tester.

**Après :**

* Car → modèle de données
* CarRepository → gestion et accès aux données
* CarFormatter → formatage et présentation
* CarEvaluator → logique métier
* Main → coordination

**Résultat :** Chaque classe a une seule responsabilité → code plus clair, modulaire et testable.

### **2️ OCP – Open/Closed Principle**

**Avant :**

* ResourceAllocator gérait différents types de ressources via un switch (TIME\_SLOT, SPACE\_SLOT).
* Chaque nouveau type nécessitait de modifier la classe principale.

**Problèmes :**

* Couplage fort
* Risques de régression
* Pas extensible

**Après :**

* Classe abstraite Resource avec allocate() et free()
* Sous-classes : TimeSlot, SpaceSlot, etc.
* ResourceAllocator délègue simplement l’appel à resource.allocate()

**Résultat :** Le code est **ouvert à l’extension**, **fermé à la modification** → extensible et stable.

### **3️ LSP – Liskov Substitution Principle**

**Avant :**

* ElectronicDuck héritait de Duck mais lançait des exceptions si elle était “off”.
* Pool utilisait Duck sans savoir si c’était un ElectronicDuck.

**Problème :** ElectronicDuck ne pouvait pas remplacer Duck sans erreurs → violation du LSP.

**Après :**

* Création de l’interface DuckBehavior (quack(), swim()).
* Duck et ElectronicDuck implémentent cette interface.
* Pool dépend de DuckBehavior.

**Résultat :** Chaque “canard” est interchangeable → substitution correcte, comportement cohérent.

### **4️ ISP – Interface Segregation Principle**

**Avant :**

* Interface Door avec toutes les méthodes : open, close, lock, unlock, timeOutCallback, proximityCallback.
* SensingDoor et TimedDoor devaient implémenter des méthodes inutiles → exceptions lancées.

**Problème :** Classes forcées d’implémenter des méthodes non utilisées → violation de l’ISP.

**Après :**

* Door → méthodes de base.
* ProximitySensitive → gestion de la proximité.
* TimedSensitive → gestion du temps.
* SensingDoor implémente Door + ProximitySensitive.
* TimedDoor implémente Door + TimedSensitive.

**Résultat :** Interfaces plus petites et spécifiques → code clair, flexible et cohérent.

### **5️ DIP – Dependency Inversion Principle**

**Avant :**

* EncodingModule lisait et écrivait directement dans des fichiers, le réseau ou la base de données.
* Connaissait tous les détails bas niveau (FileDataReader, NetworkReader, MyDatabase).

**Problème :**

* Couplage fort entre haut et bas niveau.
* Difficile à tester, pas extensible.

**Après :**

* Interfaces IDataReader et IDataWriter pour abstraction.
* EncodingModule dépend uniquement de ces interfaces.
* Implémentations concrètes : FileDataReader, FileDataWriter, NetworkDataReader, DatabaseDataWriter.

**Résultat :** Le module haut-niveau dépend d’abstractions, pas de détails concrets → code extensible, testable et maintenable.

## **Conclusion**

| **Principe** | **Problème avant** | **Solution après** | **Résultat** |
| --- | --- | --- | --- |
| **SRP** | Classe avec plusieurs responsabilités | Division en classes spécialisées | Code clair et maintenable |
| **OCP** | Modifications fréquentes du code existant | Héritage et polymorphisme | Extensible sans modifier |
| **LSP** | Sous-classes non substituables | Interface commune cohérente | Substitution sans erreurs |
| **ISP** | Interfaces trop grandes | Interfaces fines et spécifiques | Flexibilité accrue |
| **DIP** | Haut-niveau dépend des détails | Abstraction via interfaces | Couplage faible et testabilité |