

Application des principes SOLID :

Single Responsibility Principle :

```
@Override  
public void connexionBDD() {  
    ConnecteurMySql db = new ConnecteurMySql();  
    db.connect("localhost", 3306, "root", "root");  
}  
  
@Override  
public void envoyerNotifEmail(String message) {  
    EmailSmtp sender = new EmailSmtp();  
    sender.send("user@example.com", "Message de votre bibliothèque", message);  
}
```

Le principe de Single Responsability n'est pas respecter car ici, la classe Livre gère la connexion à la base de donnée et l'envoi des mails. Le problème est également présent dans DVD.

Pour régler ce problème, on utilise des classes ConnecteurMySQL et EmailSmtp qui gèrent les connexions et les mails à la place de la classe livre. On fait de même avec DVD

Open/Closed Principle :

```
public void processItem(Object item, String type) {  
    if ("livre".equalsIgnoreCase(type)) {  
        Livre l = (Livre) item;  
        l.emprunter();  
        l.imprimeEtiquette();  
        l.connexionBDD();  
        l.envoyerNotifEmail("Livre emprunté !");  
    } else if ("dvd".equalsIgnoreCase(type)) {  
        DVD d = (DVD) item;  
        d.rendreArticle();  
        d.imprimeEtiquette();  
        d.connexionBDD();  
    } else {  
        System.out.println("Type d'article inconnu : " + type);  
    }  
}
```

La méthode processItem utilise une succession de if/else. Pour ajouter un nouveau type d'objet (cassette par exemple), on devra modifier le code pour pouvoir l'intégrer.

Pour régler ce problème, il faut ouvrir à l'extension et fermer à la modification. On doit pouvoir ajouter des extensions sans modifier le code existant.

Pour cela, on définit processItem dans une nouvelle classe abstraite, mère de DVD et Livre appelée

Item, et on redéfinit processItem dans les différentes classes. Pour ajouter Cassette, on aura juste à créer une classe et à redéfinir la méthode processItem. Nous n'aurons pas à toucher au code.

Liskov Substitution Principle :

```
@Override  
public void calculerPenaliteDeRetard(int jours) {  
    if (jours < 0) {  
        throw new IllegalArgumentException("Le nombre de jours ne peut pas être négatif !");  
    }  
    System.out.println("Pénalités de retard : " + (jours * 10) + " euros");  
}
```

Le principe de substitution de Liskov n'est pas respecté car la pénalité de retard a un comportement différent en fonction du type d'objet où elle est appelée. Dans plus, elle introduit une condition de jours positifs mais renvoi la pénalité dans tout les cas.

Pour régler ce problème, j'ai crée une interface Penalité qui permet d'implémenter la méthode et de la redéfinir pour chaque Item pénalisable, et j'ai placé le calcul de la pénalité dans un else pour éviter le calcul en cas d'entrée incohérente.

Interface Segregation Principle :

```
public interface ArticleEmpruntable {  
  
    void emprunter();  
  
    void rendreArticle();  
  
    void calculerPenaliteDeRetard(int jours);  
  
    void imprimeEtiquette();  
  
    void connexionBDD();  
  
    void envoyerNotifEmail(String message);  
}
```

Ici le principe de segregation des interfaces n'est pas respecté car les articles empruntables sont forcées d'implémenter des méthodes inutiles

Pour régler ce problème, en divise notre interface en plusieurs interfaces (ArticleEmpruntable, ArticleRetournable etc.)

Dependency Inversion Principle :

```
class MySqlDatabase {  
    public void connect(String host, int port, String user, String pass) {  
        System.out.println("Connexion MySQL à " + host + ":" + port);  
    }  
}
```

Le principe d'inversion des dépendances n'est pas respecté car la classe dépend d'une classe de bas niveau.