Principes de conception orientée-objet

Principes SOLID

SRP

SRP consiste à créer une classe par fonctionnalité. Cela permet d'éviter qu'une autre classe dépende de plusieurs fonctionnalités à la fois alors qu'elle en dépend que d'une seule. Si une classe possède plusieurs responsabilités, elle aura plusieurs raisons de changer.
Il n'est pas nécessaire de découpler les responsabilités si les changements n'ont aucun risque de se produire, ou s'ils se produisent toujours ensemble. (ex : Employe, EmployeSalaire, EmployeCoordonnees).

OCP

OCP consiste à utiliser l'héritage, l'abstraction de classes et le polymorphisme pour pouvoir facilement étendre des modules mais sans devoir les modifier. (ex : Employe, Vendeur, Manager).

LSP

LSP consiste à utiliser de l'héritage pour pouvoir avoir des méthodes communes. Si une sous classe ne doit pas implémenter une méthode, il ne faut pas l'implémenter dans la classe mère. (ex : Robot, RobotStatique, RobotDynamique).

ISP

ISP consiste à utiliser des interfaces découpées en fonction des besoins et ne pas regrouper une grosse inrterface pour tous les besoins. (ex : Printer, Scanner, Copyer, Faxer comme interfaces et SimplePrinter implémente uniquement Printer).

DIP

DIP consiste à ce que les modules haut niveaux ne doivent pas dépendre de modules bas niveau. Pour cela, on crée une méthode dans la classe métier A qui prend en paramètre une interface I. On crée une classe B qui implémente l'interface I. Dans la méthode de A, on appelle donc l'interface I avec la méthode implémentée par B. (ex : Classe Metier, Interface Logger, Classe ConsoleLogger qui implémente Logger).

ex : au lieu de faire

```java
public class MetierFaux {

    private int val;


    public Metier(int val) {
        this.val=val;
    }

    public void methode(Logger logger) {
        System.out.println(LocalDateTime.now()+ ": Début de methode");
        this.val++;
        System.out.println(LocalDateTime.now() + ": Fin de methode");
    }

```

```
}
```
il faut faire :

```java
public class Metier {

    private int val;


    public Metier(int val) {
        this.val=val;
    }

    public void methode(Logger logger) {
        //logger = Logger.getLogger("logger");
        logger.log(LocalDateTime.now()+ ": Début de methode");

        this.val++;
        logger.log(LocalDateTime.now() + ": Fin de methode");
    }

}


public class ConsoleLogger implements Logger{
    public ConsoleLogger() {

    }

    public void log(String string) {
        System.out.println(string);
    }
}

public interface Logger {
    public void log(String string);
}
```

Patterns GRASP

Polymorphisme, Expert en information, Créateur, Fabrication pure, Faible couplage, Indirection, Forte cohésion, Protection, Contrôleur

Définitions

- Idiome : construction utilisée de façon récurrente dans un langage de programmation donné pour réaliser une tâche « simple » (i++ par ex)pour parcourir les éléments d'une collection
- Pattern d'architecture : solution générique et réutilisable à un problème d'architecture logicielle
- Pattern d'entreprise : solution pour la structuration d'une application d'entreprise
- Anti-pattern : solution commune à un problème récurrent mais qui est en général inefficace et contre-productive