CLEAN CODE

Objets

ABSTRACTION DE DONNÉES

Quand on parle d'abstraction, on ne veut pas seulement cacher les données, mais aussi la structure d'implémentation du code.

Par exemple, si on veut donner les coordonnées d'un point, regardons deux structures de données:

```
public class Point {
  public double x;
  public double y;
}
```

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Les **interfaces abstraites** permettent aux utilisateurs de manipuler l'**essence des données** sans connaître leur format (CartesianPoint vs PolarPoint)

Il faut **masquer l'implémentation**. Il ne s'agit pas seulement de rendre les variables internes `privées` (private), mais de ne **jamais exposer** la façon dont la donnée est structurée ou stockée.

Si plusieurs variables doivent être mises à jour ensemble pour que l'objet reste valide, cette mise à jour doit être perçue comme une **opération unique et indivisible** (**opération atomique).**

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

```
public class CartesianPoint implements Point {

    private double x;
    private double y;

    public CartesianPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public double getX() {
        return this.x;
    }

    @Override
    public double getY() {
        return this.y;
    }

    @Override
    public void setCartesian(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Continued

```
    @Override
    public double getR() {
        return Math.hypot(this.x, this.y);
    }

    @Override
    public double getTheta() {
        return Math.atan2(this.y, this.x);
    }

    @Override
    public void setPolar(double r, double
theta) {
        this.x = r * Math.cos(theta);
        this.y = r * Math.sin(theta);
    }
}
```

ABSTRACTION DE DONNÉES

Dernier exemple,

```
public interface Voiture {
  double get_GrosseurDuReservoir();
  double get_QuantiteDeGaz();
}
```

```
public interface Voiture {
  double get_PourcentageDeCarburantRestant();
}
```

Ici, bien que les deux méthodes soient publiques, l'une révèle trop d'informations sur le véhicule. Il faut favoriser l'abstraction à l'implémentation.
L'interface devrait exposer des **concepts abstraits et universels** liés à l'**autonomie** et au **niveau d'énergie**, qui fonctionnent pour **tous** les types de véhicules.

Objets VS Structures de données

Les **objets** permettent de cacher les données derrière de l'abstraction et expose les fonctions qui opèrent sur les données. Ce sont des conteneurs passifs.
Les **structures de données** exposent leurs données et ne contiennent pas de logique métier.
Ils sont un peu à l'opposée.

```
class PointRectangulaire:
    def __init__(self, x, y):
        self.coordonnee_x = x
        self.coordonnee_y = y

def toString(self): return f"Coordonnées
Rectangulaires: ({self.x}, {self.y})"
```

```
class Point:
    def __init__(self, x, y):
        self._coordonnee_x = x
        self._coordonnee_y = y

    def deplacer(self, delta_x, delta_y):
        self._coordonnee_x += delta_x
        self._coordonnee_y += delta_y
```

```
public class Square {
	public Point topLeft;
	public double side;
}
public class Circle {
	public Point center;
	public double radius;
}
public class Geometry {
	public double area(Object shape){
		if (shape instanceof Square) {
			Square s = (Square)shape;
			return s.side * s.side;
		}
		else if (shape instanceof Circle) {
			Circle c = (Circle)shape;
			return PI * c.radius * c.radius;
		}
	}
}
```

Ce code est structuré comme une structure de données (style procédural).

Le code procédural est **flexible** face à l'ajout de nouvelles **fonctions**, mais **rigide** face à l'ajout de nouveaux **types de données**.

```
public interface Forme {
    double calculerAire();
    double calculerPerimetre();
}

public class Carre implements Forme {
    private double cote;

    public Carre(double cote) {
        this.cote = cote;
    }

    @Override
    public double calculerAire() {
        return this.cote * this.cote;
    }

    @Override
    public double calculerPerimetre(){
        return 4 * this.cote;
    }
}
```

Ce code est structuré selon le modèle d'objet (style OO).

Le modèle POO est **rigide** face à l'ajout de nouvelles **fonctions**, mais **flexible** face à l'ajout de nouveaux **types de données**.

Ceci illustre l'anti-symétrie entre un objet et une structure de données.

Il faut déterminer quand il est plus juste d'utiliser l'un ou l'autre, en anticipant l'évolution du code.

Aussi, on veut un **OU** l'autre, jamais un hybride chancelant.

LOI DE DÉMÉTER

La loi de Déméter stipule qu'un objet ne devrait interagir qu'avec ses relations directes, **pas avec les objets internes des autres**.

Par exemple:

```
order.getClient().getAddresse().getVille()
```

Le code connait trop d'information (client, adresse, ville)
Si la fonction Customer u Adresse change, la fonction getVille peut faire briser le code.
Donc, le code est fragile et difficile à maintenir.

Un meilleur exemple serait simple

```
order.getCustomerCity()
```

On appelle parfois cette chaine de méthode le «Train Wreck», comme toutes les méthodes se suivent comme les wagons d'un train, mais si un seul déraille….

LOI DE DÉMÉTER

De manière simplifiée:

Un objet A peut parler à :
1. Lui-même
2. Ses attributs directs
3. Les paramètres reçus
4. Les objets qu'il crée directement

Et pas à l'objet de l'objet de l'objet

```
class Client:
    def envoyer_facture(self): print("Facture envoyée")  # 2. Attribut direct
    @property
    def ville(self): return "Montréal"                   # 3. Paramètre reçu

class Facture:
    def __init__(self, client): self.client = client
    def envoyer(self): print("Facture traitée")          # 4. Objet créé directement

class Commande:
    def __init__(self, client): self.client = client
    def traiter(self, cmd_externe):
        self.calculer_total()                            # 1. Lui-même
        self.client.envoyer_facture()                    # 2. Attribut direct
        print(cmd_externe.client.ville)                  # 3. Paramètre reçu
        Facture(self.client).envoyer()                   # 4. Objet créé directement
    def calculer_total(self): print("Total calculé")
```

Mais attention, une chaine de commande peut être acceptable, surtout s'il s'agit d'une lecture simple d'attribut ou un path.

```
String ville = client.adresse.ville;
```

Comparé à une chaine de méthodes.

```
String path = ctxt.getClient().getAdresse().getVille();
```

Les **principes SOLID** sont des directives essentielles pour la programmation orientée objet, visant à créer un code propre et maintenable. Les voici :

- **S** : Single Responsibility Principle (SRP) - Une classe doit avoir une seule responsabilité.
- **O** : Open/Closed Principle (OCP) - Les entités doivent être ouvertes à l'extension mais fermées à la modification.
- **L** : Liskov Substitution Principle (LSP) - Les objets d'une classe dérivée doivent pouvoir remplacer ceux de la classe de base sans altérer le fonctionnement du programme.
- **I** : Interface Segregation Principle (ISP) - Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.
- **D** : Dependency Inversion Principle (DIP) - Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais des abstractions.

SRP - Single Responsability Principle

```
// Responsabilité 1: Contient la logique métier des données
public class DonnéesRapport {
    private List<String> donnees;
    // ...
}

// Responsabilité 2: Gère uniquement la présentation
public class FormatageHTML {
    public String formater(DonnéesRapport rapport) {
        return "<html>...</html>";
    }
}

// Responsabilité 3: Gère uniquement l'entrée/sortie (I/O)
public class PersistanceDB {
    public void sauvegarder(DonnéesRapport rapport) {
        // Logique de sauvegarde
    }
}
```

Principe

Une classe ne doit avoir qu'**une seule raison d'être modifiée**.

Cela signifie qu'elle ne doit gérer qu'**une seule responsabilité métier**.

OCP - Open-Closed Principle

```
public interface Employe {
    double calculerSalaire();
}

public class EmployeContractuel implements Employe {
    @Override
    public double calculerSalaire() {
        // Logique contractuelle
        return 50000.0;
    }
}

public class CalculateurSalaire {
    public double calculerSalaireTotal(List<Employe> employes) {
        double total = 0.0;
        for (Employe e : employes) {
            // faire confiance à l'objet de connaître sa propre logique
            total += e.calculerSalaire();
        }
        return total;
    }
}
```

Principe

Les entités logicielles (classes, modules, fonctions, etc.) doivent être **ouvertes à l'extension** (pour ajouter de nouveaux comportements) mais **fermées à la modification** (pour éviter de toucher au code existant).

LSP - Liskov Substitution Principle

```
public class Rectangle {
    protected double largeur, hauteur;
    // Un invariant implicite: on peut changer l'un sans changer l'autre
    public void setLargeur(double l) { this.largeur = l; }
    public void setHauteur(double h) { this.hauteur = h; }
}

public class Carre extends Rectangle {
    // La substitution est brisée
    @Override
    public void setLargeur(double l) {
        this.largeur = l;
        this.hauteur = l; //Effet de bord
    }
    // ... même chose pour setHauteur ...
}
```

Principe

Les objets d'une classe dérivée doivent pouvoir **remplacer** (être substitués par) ceux de la classe de base sans altérer le fonctionnement du programme.

Dans ce contre-exemple, La sous-classe Carré modifie l'invariant de la classe parente Rectangle (l'indépendance de la largeur et de la hauteur), ce qui peut briser le code client.

ISP - Interface Segregation Principle

Principe: Les clients ne doivent pas être obligés de dépendre d'interfaces qu'ils **n'utilisent pas**. Les interfaces doivent être **petites** et **spécifiques** au rôle.

```
// Interface "gonflée"
public interface MachineMultiFonction {
    void imprimer();
    void numériser();
    void faxer();
}

public class ImprimanteSimple implements MachineMultiFonction {
    @Override public void imprimer() { /* Imprime bien */ }
    @Override public void numériser() { /* Scan bien */ }
    @Override public void faxer() {
        throw new UnsupportedOperationException();
    }
}
```

```
public interface Imprimeur {
    void imprimer();
}

public interface Faxeur {
    void faxer();
}

public interface Numériseur {
    void numériser();
}

// La classe implémente uniquement ce dont elle a besoin
public class ImprimanteSimple implements Imprimeur {
    @Override public void imprimer() { /* Imprime bien */ }
}
```

DIP - Dependency Inversion Principle

Principe: Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'**abstractions** (interfaces).

```
// Module de haut niveau (Logique métier)
public class ServiceNotifications {
    // Dépendance CONCRÈTE à une classe de bas niveau
    private EmailSender emetteur = new EmailSender();

    public void envoyerAlerte(String message) {
        emetteur.envoyer(message);
    }
}

// Module de bas niveau (Détail technique)
public class EmailSender {
    public void envoyer(String msg) { /* Envoie le mail */ }
}
```

```
public interface Notificateur {
    void notifier(String message);
}

public class EmailSender implements Notificateur {
    @Override
    public void notifier(String msg) { /* Envoie le mail */ }
}

public class ServiceNotifications {
    private final Notificateur notificateur;

    public ServiceNotifications(Notificateur notificateur) {
        this.notificateur = notificateur;
    }

    public void envoyerAlerte(String message) {
        notificateur.notifier(message);
    }
}
```