

# UNIVERSITE DE CORSE 2024-2025

## Licence ST 3ème année Option INFORMATIQUE

### UE Qualité Logicielle et Tests CH3 –Bonnes pratiques OO Principes SOLID



Evelyne VITTORI  
[vittori@univ-corse.fr](mailto:vittori@univ-corse.fr)

# Plan du cours



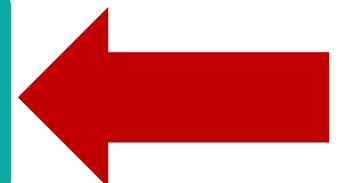
CH1 – Principes de Qualité



CH2 – Métriques logicielles



CH3 - Bonnes pratiques OO



CH4 – Tests

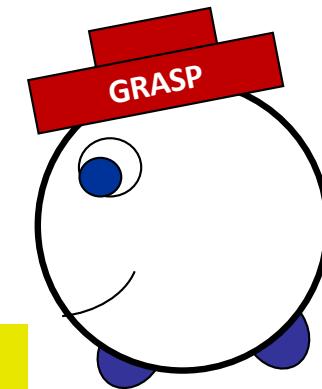
# Objectifs de ce chapitre

- Etude des principes de base de conception OO

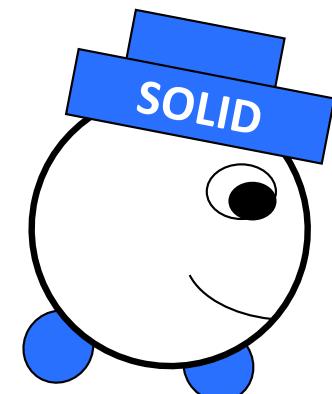


- Patterns GRASP

Non étudiés en cours  
cette année  
(diapos sur l'ENT pour  
information)



- Principes SOLID





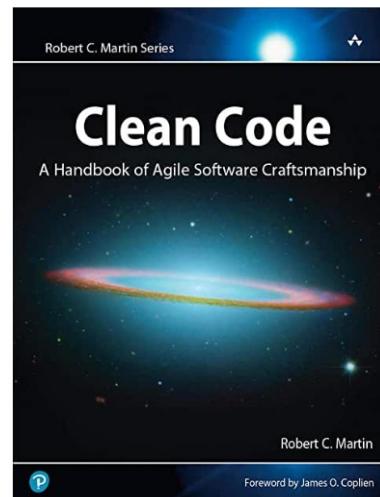
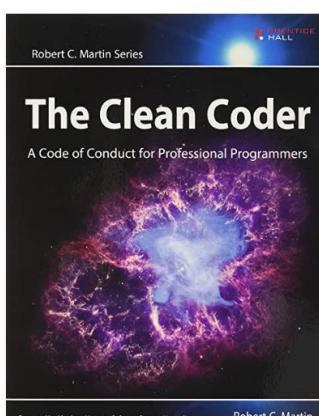
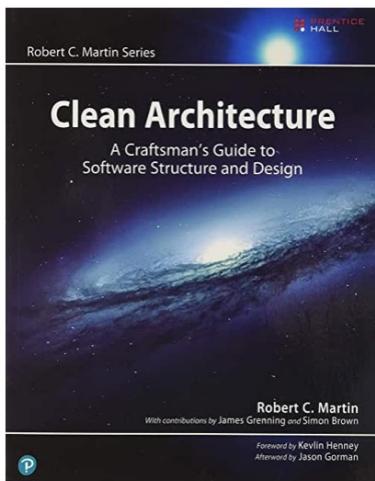
## CH3 –Bonnes pratiques OO

# Principes SOLID



# Principes SOLID

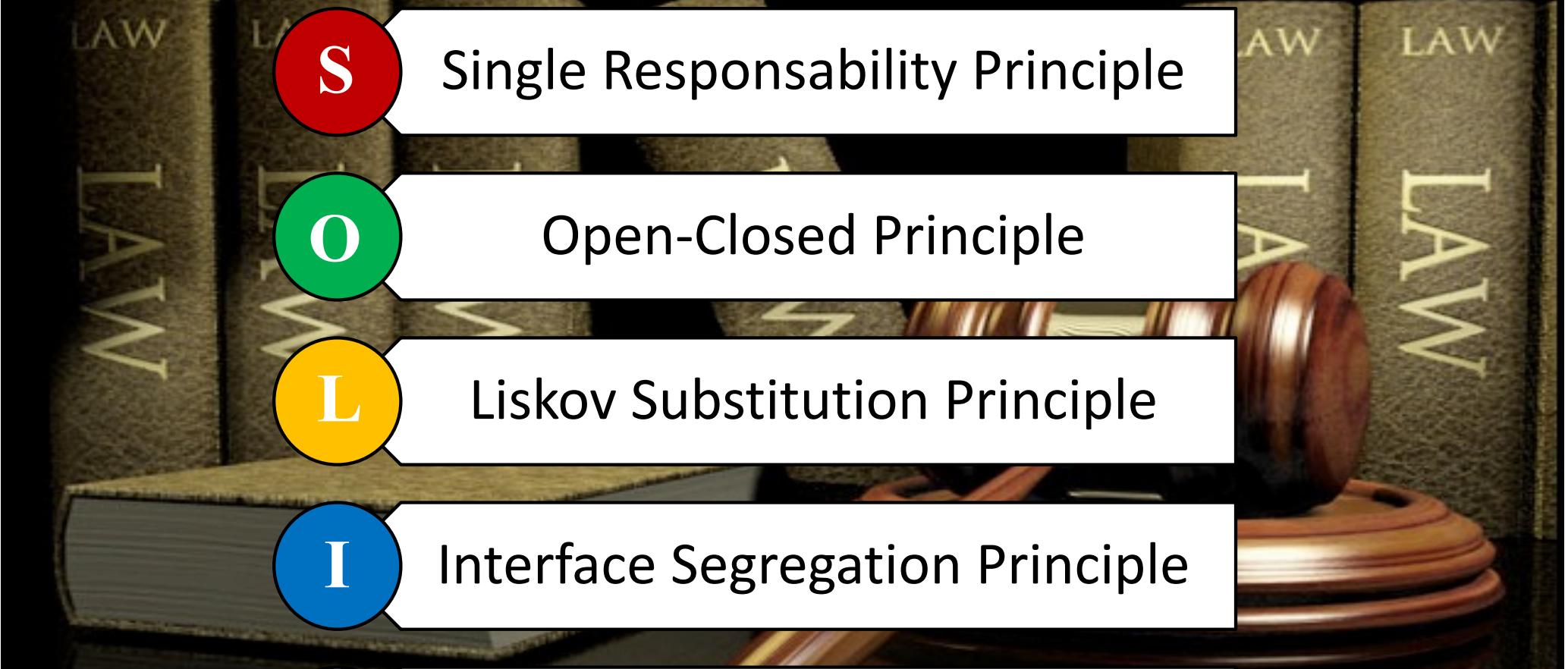
- Acronyme proposé par Michael Feathers pour désigner 5 principes de conception identifiés par « Oncle Bob » en 2000.
- Principes visant à concevoir des logiciels :
  - plus compréhensibles
  - plus faciles à maintenir
  - plus faciles à faire évoluer



Oncle BOB  
**(Robert Cecil Martin)**



# Principles SOLID



S

Single Responsibility Principle

O

Open-Closed Principle

L

Liskov Substitution Principle

I

Interface Segregation Principle

D

Dependency Inversion Principle



# Principe de responsabilité unique



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

*Ce n'est pas parce que vous le pouvez  
que vous devez le faire!*



# Principe de responsabilité unique

- Une classe doit faire une seule chose mais le faire bien.
- Exemple Site Web marchand
  - Classe **GestionPanier**
    - Responsabilité =gestion du panier
    - Méthodes= ajouter un produit, modifier un produit, supprimer un produit du panier
    - Autres responsabilités qui doivent être placées ailleurs = validation de la commande, paiement, ...



# Principe de responsabilité unique

Pourquoi?

- Trop de responsabilités tue!



# Ex: Classe ne respectant pas SRP

```
public class Database {
    ArrayList<Product> productsInMemory;
    public void connectBD()
    {
        // connexion à la base de données
    }
    public void loadData()
    {
        //Chargement des données à partir de la Base de données
        productsInMemory = new ArrayList<Product>();
        productsInMemory.add( new Product ("Product1", 100));
        productsInMemory.add( new Product ("Product2", 255));
    }
    public ArrayList<Product> getProductsTTC()
    {
        connectBD();
        loadData();
        // Calcul des prix TTC
        ArrayList<Product> res= new ArrayList<Product>();
        //return liste des produits TTC;
        return res;
    }
}
```

Classe non cohésive : 3 responsabilités

- Connexion BD
- Chargement de données
- Calcul TVA



```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name=name;
        this.price=price;
    }
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
}
```

# Ex: Refactoring pour appliquer SRP

```
public class DatabaseConnection {  
  
    public void connectBD()  
    {  
        // connexion à la base de données  
    }  
}
```

- La classe Database est décomposée en 2 nouvelles classes
- La responsabilité du calcul du prix TTC est donnée à product

```
public class ProductRepository {  
    DatabaseConnection mabd;  
    ArrayList<Product> productsInMemory;  
  
    public ProductRepository()  
    {  
        mabd=new DatabaseConnection();  
    }  
  
    public void loadData()  
    {  
        //Chargement des données à partir de la base de données  
        productsInMemory = new ArrayList<Product>();  
        productsInMemory.add( new Product ("Produit 1", 10.0) );  
        productsInMemory.add( new Product ("Produit 2", 20.0) );  
    }  
}
```

```
public class Product {  
    private String name;  
    private double price;  
    public final static double VATRate=0.196;  
  
    public Product(String name, double price) {  
        this.name=name;  
        this.price=price;  
    }  
    public String getName() {  
        return name;  
    }  
    public double getPrice() {  
        return price;  
    }  
    public double getTTCPrice()  
    {  
        return price * (1 + VATRate);  
    }  
}
```





# Principe de responsabilité unique

## Pour les méthodes aussi!

Décomposer, séparer les responsabilités

- Pour faciliter la maintenance
- Pour augmenter la réutilisabilité

### Principe de séparation Commande-Requête

Une méthode doit être une « **commande** » ou une « **requête** » mais jamais les deux à la fois.

- Méthodes « **commande** » : effectuent des actions susceptibles de modifier l'état d'un objet (attributs) et n'ont pas de valeur de retour (*void* en Java).
- Méthodes « **requête** » : renvoient des données mais n'effectuent aucune modification de l'état d'un objet.

Bertrand Meyer  
1988



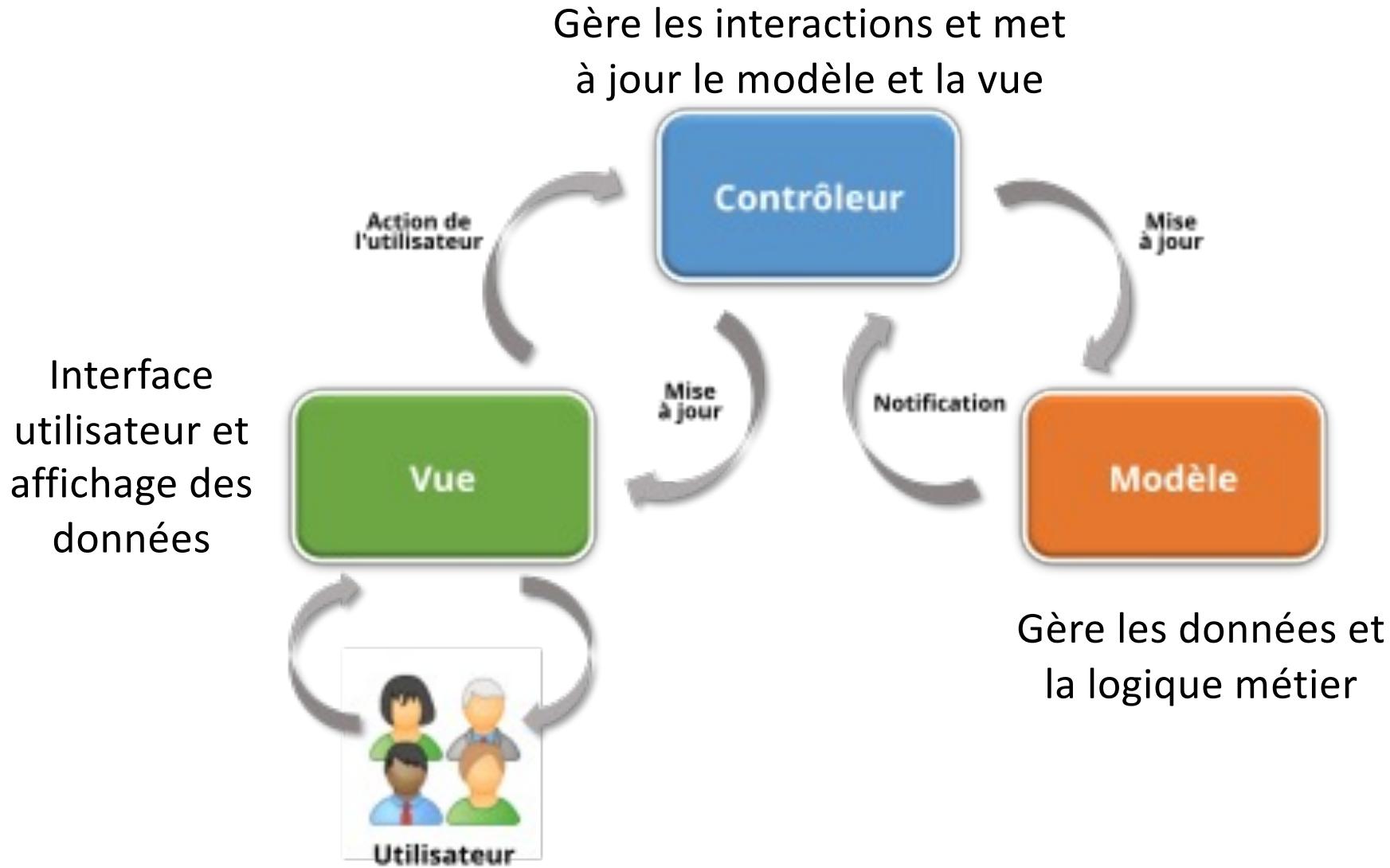
# Le modèle MVC

## Un exemple d'application du principe SRP

- Le modèle MVC (Modèle-Vue-Contrôleur) est une architecture logicielle qui sépare une application en 3 parties distinctes
- Objectifs
  - modularité, testabilité et maintenance du code
- En particulier, gestion facilitée des changements d'interface utilisateur

Plusieurs variantes

# Pattern MVC classique



# Le Modèle

- Gère l'accès aux données
  - fichiers, base de données, objets en mémoire
- Implémente la logique métier
- Ne s'occupe ni de l'affichage ni des interactions avec l'utilisateur

```
// Modèle : Stocke les données
class ModeleUtilisateur {
    private String nom;
    public Utilisateur(String nom) { this.nom = nom; }
    public String getNom() { return nom; }
}
```

Gère les interactions avec les bases de données

# La Vue

- Affiche les données
- Reçoit les entrées utilisateur
- Ne communique qu'avec le contrôleur

```
// Vue : Interface utilisateur en console
class VueUtilisateur {
    private Scanner scanner;
    private ControleurUtilisateur controleur;

    public VueUtilisateur() {
        this.scanner = new Scanner(System.in);
    }

    public void setControleur(ControleurUtilisateur controleur) {
        this.controleur = controleur;
    }

    public void saisirUtilisateur() {
        System.out.print("Entrez le nom de l'utilisateur : ");
        String nom = scanner.nextLine();
        controleur.creerUtilisateur(nom);
    }

    public void afficherUtilisateur(String nom) {
        System.out.println("Nom de l'utilisateur : " + nom);
    }
}
```

# Le contrôleur

- Met à jour le modèle et la vue en conséquence
- Interagit entre le modèle et la vue

```
// Contrôleur : Intermédiaire exclusif entre Vue et Modèle
class ControleurUtilisateur {
    private ModeleUtilisateur utilisateur;
    private VueUtilisateur vue;

    public ControleurUtilisateur(VueUtilisateur vue) {
        this.vue = vue;
        this.vue.setControleur(this);
    }

    public void creerUtilisateur(String nom) {
        utilisateur = new Utilisateur(nom);
        mettreAJourVue();
    }

    private void mettreAJourVue() {
        vue.afficherUtilisateur(utilisateur.getNom());
    }
}
```

# Programme principal

- Exemple avec Vue CLI (Command Line interface)

```
// Programme principal
public class Main {
    public static void main(String[] args) {
        VueUtilisateur vue = new VueUtilisateur();
        ControleurUtilisateur controleur = new ControleurUtilisateur(vue);

        vue.saisirUtilisateur(); // Lancement du processus de saisie et d'affichage
    }
}
```

# La Vue

## (version GUI - Graphical User Interface)

- Le modèle et le contrôleur sont inchangés

```
// Vue : Interface utilisateur graphique
class VueUtilisateur extends JFrame {
    private JTextField nomField;
    private JButton submitButton;
    private JLabel resultLabel;
    private ControleurUtilisateur controleur;

    public VueUtilisateur() {
        setTitle("Saisie Utilisateur");
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        nomField = new JTextField(20);
        submitButton = new JButton("Valider");
        resultLabel = new JLabel(" ");

        add(new JLabel("Entrez le nom de l'utilisateur :"));
        add(nomField);
        add(submitButton);
        add(resultLabel);

        submitButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (controleur != null) {
                    controleur.creerUtilisateur(nomField.getText());
                }
            }
        });
    }

    public void setControleur(ControleurUtilisateur controleur) {
        this.controleur = controleur;
    }

    public void afficherUtilisateur(String nom) {
        resultLabel.setText("Nom de l'utilisateur : " + nom);
    }
}
```

# Programme principal

- Exemple avec Vue CLI (Command Line interface)

```
// Programme principal
public class Main {
    public static void main(String[] args) {
        VueUtilisateur vue = new VueUtilisateur();
        ControleurUtilisateur contrroleur = new ControleurUtilisateur(vue);

        vue.saisirUtilisateur(); // Lancement du processus de saisie et d'affichage
    }
}
```

- Exemple avec Vue GUI (Graphical User Interface)

```
// Programme principal
public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            VueUtilisateur vue = new VueUtilisateur();
            ControleurUtilisateur contrroleur = new ControleurUtilisateur(vue);
            vue.setVisible(true);
        });
    }
}
```

# Bilan Pattern MVC

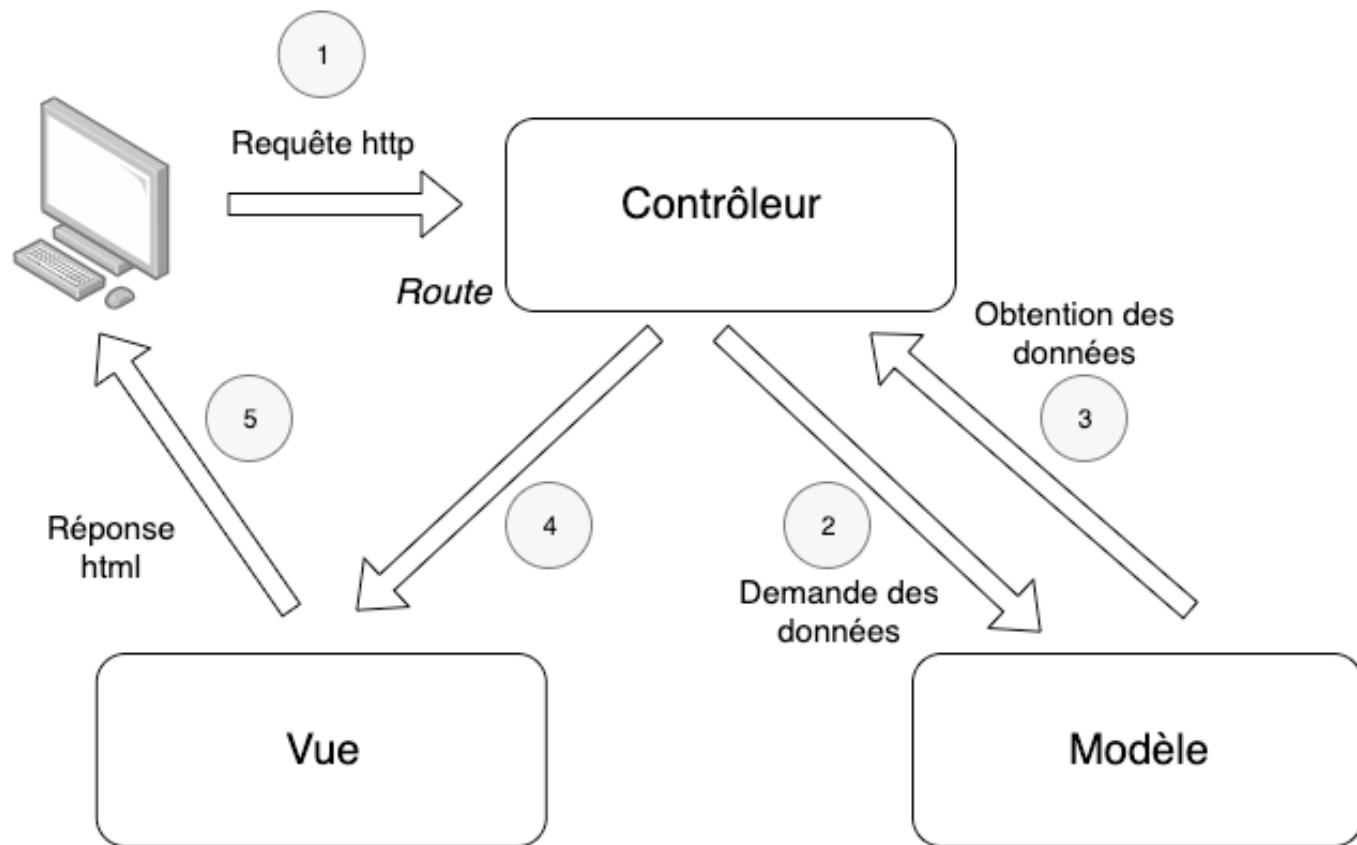
## Avantages

- Séparation des responsabilités
- Réutilisation du code
- Meilleure évolutivité

## Inconvénients

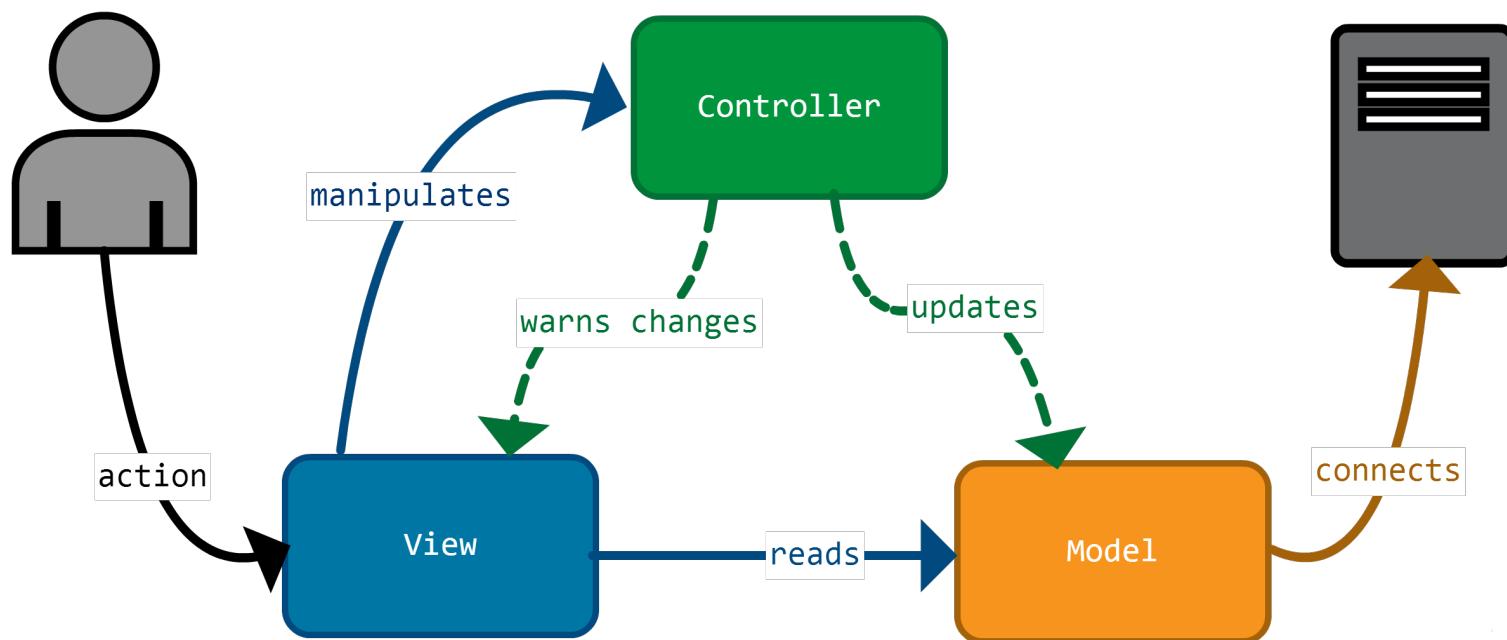
- Courbe d'apprentissage
- Complexité accrue pour les petits projets

# MVC (cas particulier Web)



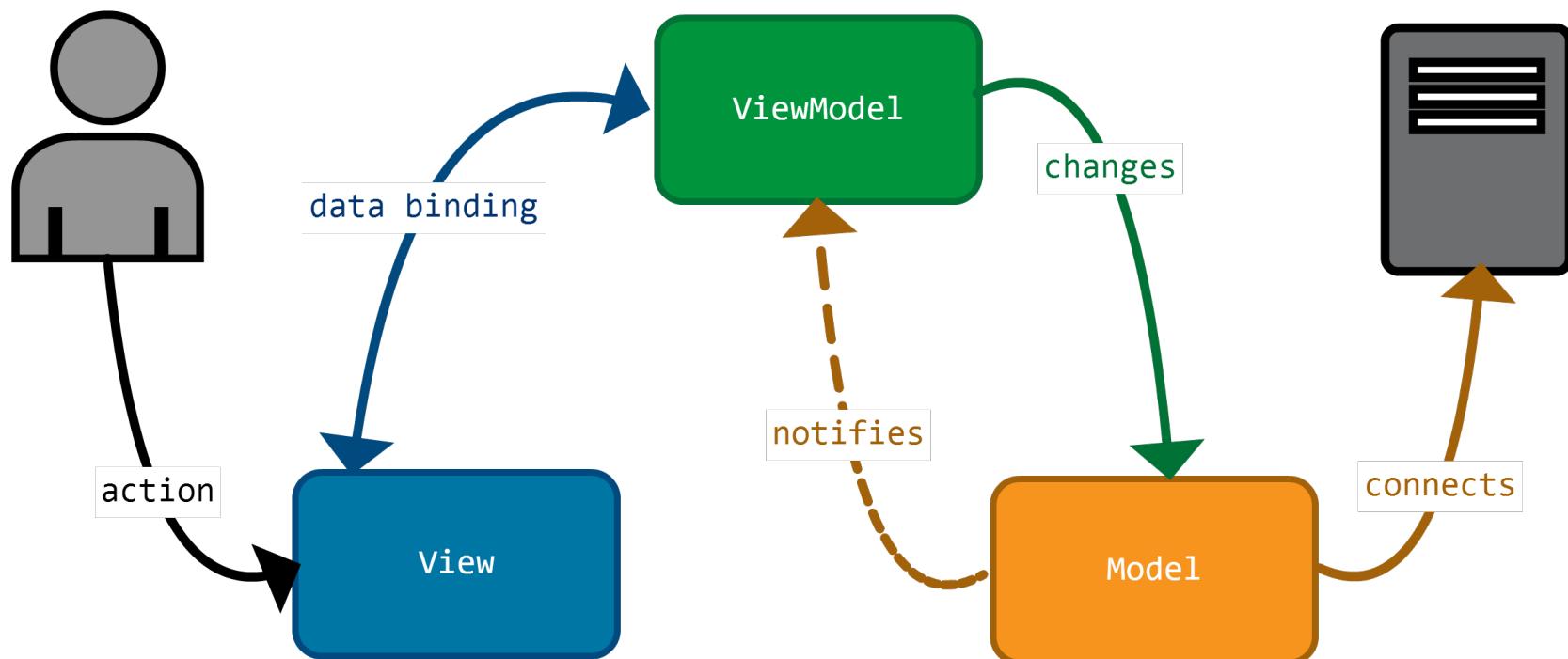
# Variante avec couplage Vue-Modele

- La Vue communique directement avec le modèle pour charger les données lorsque le contrôleur lui indique une modification



# Evolution: Pattern MVVM

- Le contrôleur est remplacé par une ViewModel
- Les Mises à jour sont automatiques (dataBinding)



# Qu'est-ce que le databinding ?

- Mécanisme qui consiste à **lier automatiquement** la vue aux données
  - Lorsqu'une donnée change dans le **Modèle**, l'affichage dans la **Vue** est **mis à jour automatiquement**, et inversement si l'utilisateur modifie un champ.



## Exemple simple :

- Un champ de texte affichant le nom d'un utilisateur peut être lié directement à la propriété nom du modèle en passant par le ViewModel.
- Toute modification du nom sera immédiatement répercutée dans l'UI sans code supplémentaire.

# Exemple de ViewModel en MVVM

- Ajout d'une propriété réactive
- Le ViewModel n'est plus dépendant de la vue

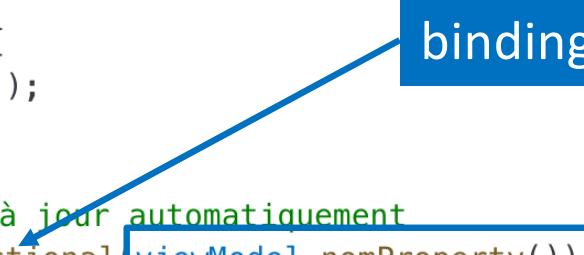
```
class UtilisateurViewModel {  
    private ModeleUtilisateur utilisateur;  
    private StringProperty nom = new SimpleStringProperty("");  
  
    public UtilisateurViewModel() {  
        this.utilisateur = new ModeleUtilisateur(""); // Création du modèle  
        nom.addListener((observable, oldValue, newValue) -> {  
            utilisateur.setNom(newValue); // ♦ Mise à jour automatique du Modèle  
        });  
    }  
  
    public StringProperty nomProperty() { return nom; }  
    public String getNom() { return nom.get(); }  
    public void setNom(String valeur) { nom.set(valeur); }  
}
```

propriété réactive

# Vue avec JavaFX (version MVVM)

- La Vue écoute directement nomProperty()

```
public class VueUtilisateur extends Application {  
    private UtilisateurViewModel viewModel = new UtilisateurViewModel();  
  
    @Override  
    public void start(Stage primaryStage) {  
        TextField nomField = new TextField();  
        Label resultLabel = new Label();  
  
        // ♦ Data Binding : la Vue se met à jour automatiquement  
        nomField.textProperty().bindBidirectional(viewModel.nomProperty());  
        resultLabel.textProperty().bind(viewModel.nomProperty());  
  
        VBox root = new VBox(10, nomField, resultLabel);  
        Scene scene = new Scene(root, 300, 200);  
  
        primaryStage.setTitle("Saisie Utilisateur - MVVM");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



binding

# Différences avec MVC classique

Aspect	MVC Classique	MVVM avec JavaFX
Qui met à jour l'UI ?	🔥 Contrôleur appelle <code>afficherUtilisateur()</code>	<input checked="" type="checkbox"/> Vue écoute <code>nomProperty()</code>
Qui gère l'état ?	🔥 Contrôleur stocke l'état de <code>Utilisateur</code>	<input checked="" type="checkbox"/> ViewModel expose <code>StringProperty</code>
Couplage Vue ↔ Contrôleur/ViewModel	🔥 Couplage fort	<input checked="" type="checkbox"/> Couplage faible (Vue seulement abonnée)
Gestion des événements utilisateur	🔥 Vue appelle Contrôleur	<input checked="" type="checkbox"/> Vue met directement à jour le ViewModel
Data Binding ?	✗ Non	<input checked="" type="checkbox"/> Oui ( <code>bindBidirectional()</code> )
Réactivité ?	✗ Non (manuelle)	<input checked="" type="checkbox"/> Oui (automatique)

# Exercice= SRP



Récupérez sur l'ENT le code java des classes Jeu et Test.

- Expliquez pourquoi ce code ne respecte pas le principe SRP au niveau de la classe Jeu et de certaines méthodes
- Effectuez un refactoring de la classe Jeu afin d'appliquer le principe SRP.
- Dans le cadre de ce refactoring, vous appliquerez le pattern MVC classique
  - Isolation des affichages dans une classe IHM spécifique
  - Création d'un contrôleur
    - principe GRASP Controleur.





# Principe Ouvert/Fermé



## Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

*Vous n'avez pas besoin de recâbler votre carte mère  
pour brancher « Mr Happy »*

# Principe Ouvert-Fermé

You should be able to extend a classes behavior, without modifying it.

Robert C. Martin.

Les classes doivent être à la fois :

- **ouvertes** pour l'extension:  
adaptables
- **fermées** à la modification: on ne doit pas avoir à modifier leur code pour les faire évoluer.

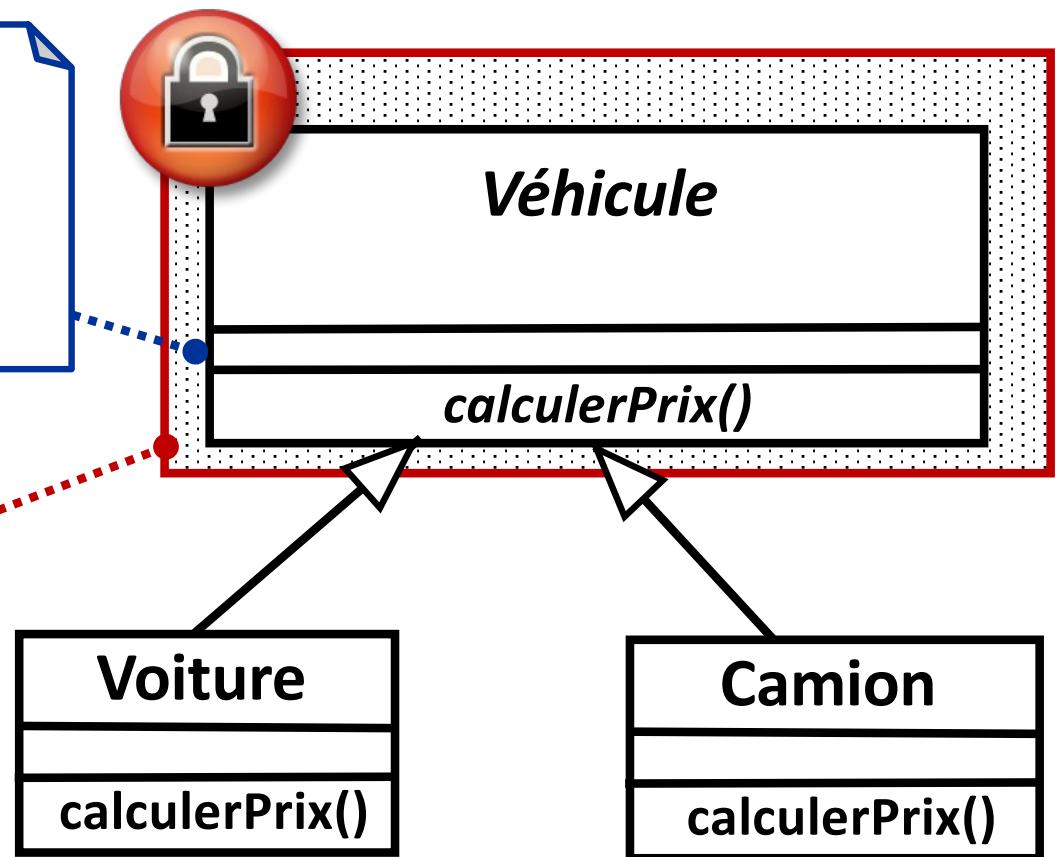




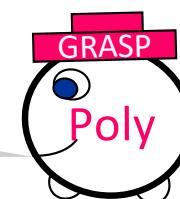
# Principe Ouvert-Fermé

**Classe fermée car on ne doit pas avoir à modifier son code pour faire évoluer l'application**

**Classe ouverte car on peut étendre son comportement en ajoutant des sous-classes (par exemple)**



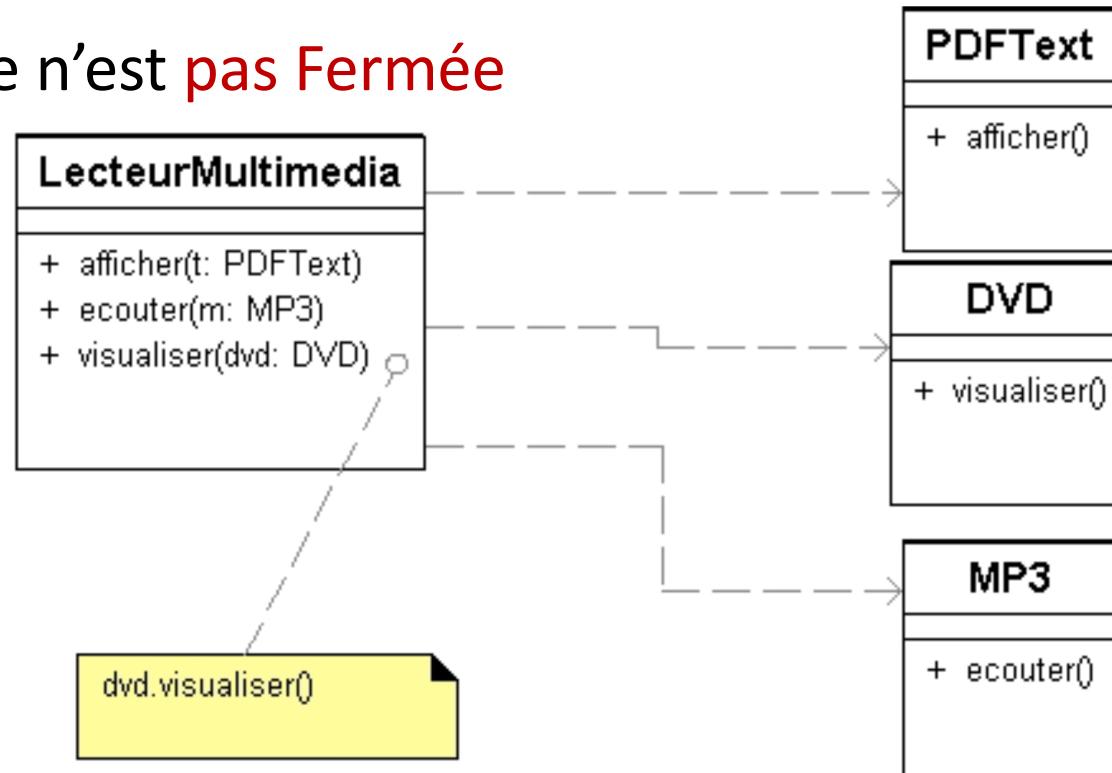
*Je reconnais mon influence!*





# Exemple violation OCP

- La classe LecteurMultimédia viole OCP
  - Nouveau type de média (mp4) ?
    - Le code de la classe LecteurMultimédia doit obligatoirement être modifié
    - La classe n'est **pas Fermée**



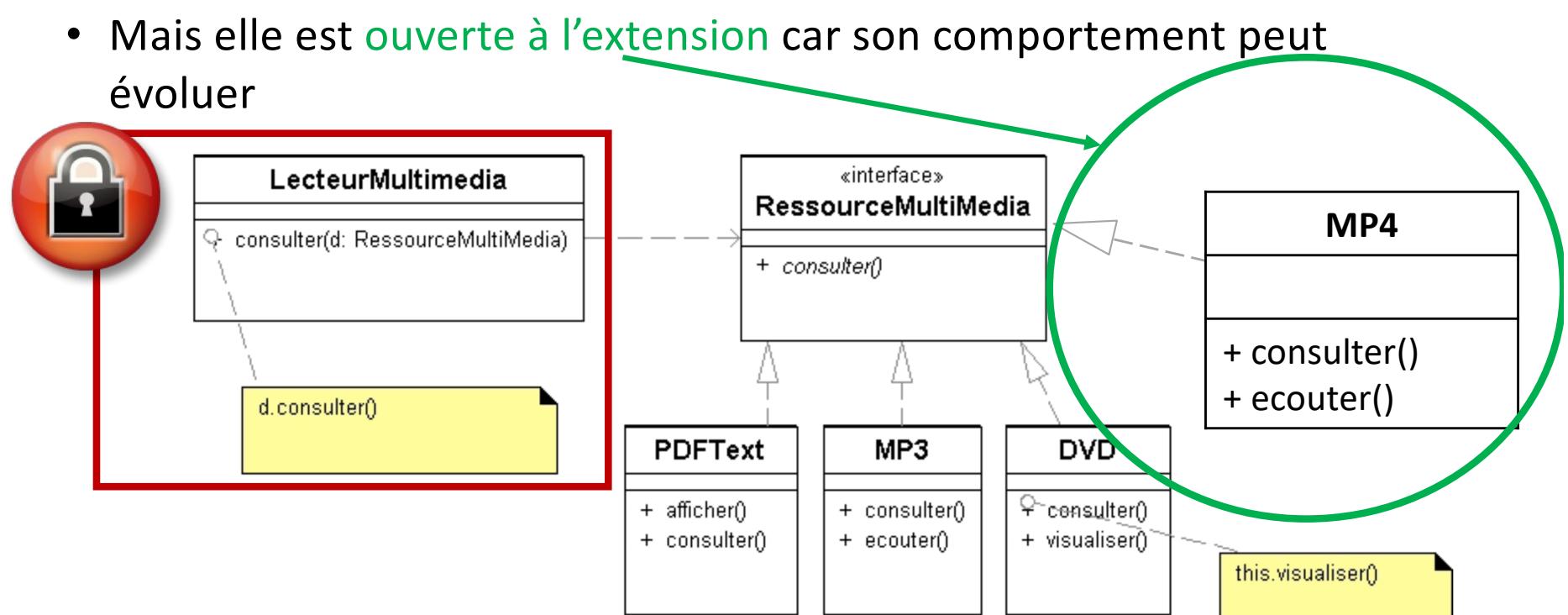


# Exemple OCP

- **Solution:** LecteurMultimédia ne doit pas dépendre des classes concrètes

- Nouveau type de média (mp4) ?

- LecteurMultimédia n'a pas à être modifiée pour évoluer: elle est **fermée à la modification**
- Mais elle est **ouverte à l'extension** car son comportement peut évoluer



# Exercice= OCP 1

La classe **Calculator** ne respecte pas OCP.

- Expliquez pourquoi.
- Proposez solution pour y remédier.
- En vous basant sur un exemple, illustrez les avantages de votre solution.

```
public class Calculator {  
    public void calculate(CalculatorOperation operation) {  
        if (operation == null) {  
            throw new InvalidParameterException("Can not perform operation");  
        if (operation instanceof Addition) {  
            Addition addition = (Addition) operation;  
            addition.setResult(addition.getLeft() + addition.getRight());}  
        else if (operation instanceof Subtraction) {  
            Subtraction subtraction = (Subtraction) operation;  
            subtraction.setResult(subtraction.getLeft() - subtraction.getRight());  
        }}  
}
```

```
public interface CalculatorOperation {  
}
```

```
public class Addition implements CalculatorOperation {  
    private double left;  
    private double right;  
    private double result = 0.0;  
  
    public Addition(double left, double right) {  
        this.left = left;  
        this.right = right;  
    }  
    // getters and setters
```

```
public class Subtraction implements CalculatorOperation {  
    private double left;  
    private double right;  
    private double result = 0.0;  
  
    public Subtraction(double left, double right) {  
        this.left = left;  
        this.right = right;  
    }  
    // getters and setters
```



# Exercice= OCP 1

## correction

```
public interface CalculatorOperation {  
    public void execOp();  
    public double getResult();  
}
```



```
public class Calculator {  
    public void calculate(CalculatorOperation operation) {  
        if (operation == null) {  
            throw new InvalidParameterException("Can not perform operation"); }  
        else {  
            operation.execOp();  
        }  
    }  
}
```

```
public class Subtraction extends Operation implements CalculatorOperation {  
  
    public Subtraction(double left, double right) {  
        super(left,right);  
    }  
  
    public void execOp() {  
        this.setResult(this.getLeft() - this.getRight());  
    }  
}
```

```
public class Addition extends Operation implements CalculatorOperation {  
  
    public void execOp() {  
        this.setResult(this.getLeft() + this.getRight());  
    }  
    public Addition(double left, double right) {  
        super(left,right);  
    }  
}
```

```
public abstract class Operation {  
    private double left;  
    private double right;  
    private double result = 0.0;  
  
    public Operation(double left, double right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    // getters and setters  
    public double getLeft() {  
        return left;  
    }  
  
    public void setLeft(double left) {  
        this.left = left;  
    }  
  
    public double getRight() {  
        return right;  
    }  
  
    public void setRight(double right) {  
        this.right = right;  
    }  
  
    public double getResult() {  
        return result;  
    }  
  
    public void setResult(double result) {  
        this.result = result;  
    }  
}
```



# Exercice= OCP 2

La classe **EventHandler** ne respecte pas OCP.

Expliquez pourquoi et proposez solution pour y remédier (**sans modifier la classe EventInterceptor**).

```
public class EventHandler {  
    private Object entity;  
    private EventInterceptor ei;  
  
    public void handleEvent(Event event) {  
        if (event instanceof PreLoad) {  
            ei.preLoad(entity);  
        } else if (event instanceof PostLoad) {  
            ei.postLoad(entity);  
        } else if (event instanceof PrePersist) {  
            ei.prePersist(entity);  
        } else if (event instanceof PreSave) {  
            ei.preSave(entity);  
        } else if (event instanceof PostPersist) {  
            ei.postPersist(entity);  
        }  
    }  
}
```

```
public class EventInterceptor {  
    public void preLoad(Object entity) {  
    }  
    public void postLoad(Object entity) {  
    }  
    public void prePersist(Object entity) {  
    }  
    public void preSave(Object entity) {  
    }  
    public void postPersist(Object entity) {  
    }  
}
```



Indique très souvent une mauvaise conception



```
abstract class Event{
    public abstract void treatment(Object o,EventInterceptor ei);

class Preload extends Event{
    public void treatment(Object o, EventInterceptor ei){
        ei.preload(o);}

//dans la méthode handleEvent
public void handleEvent(Event event){
    event.treatment(entity, ei);
```

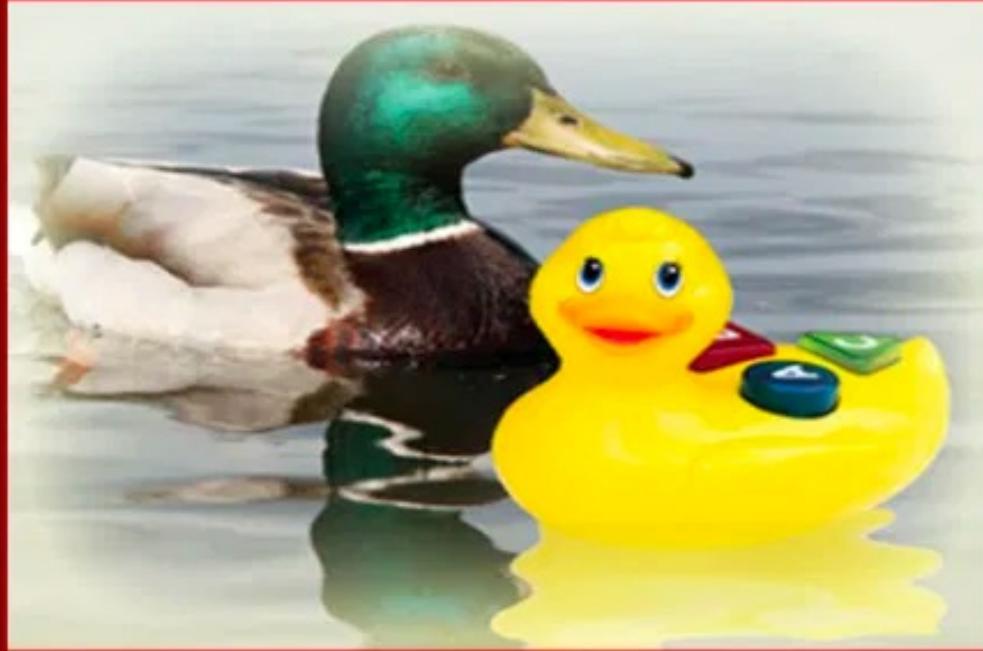


# Principe Ouvert-Fermé

- OCP améliore de manière très significative la maintenabilité et l'évolutivité.
- Toutes les classes ne peuvent pas satisfaire OCP :
  - l'objectif doit être de minimiser le nombre de classes non OCP.
  - Regrouper le maximum de code « pérenne » dans les classes satisfaisant OCP.



# Principe de substitution de Liskov



## Liskov Substitution Principle

If it looks like a DUCK, quacks like a DUCK, *but* needs BATTERIES  
- You probably need a better Abstraction -

*On ne peut pas substituer un canard en plastique à un vrai canard, c'est trop différent!*

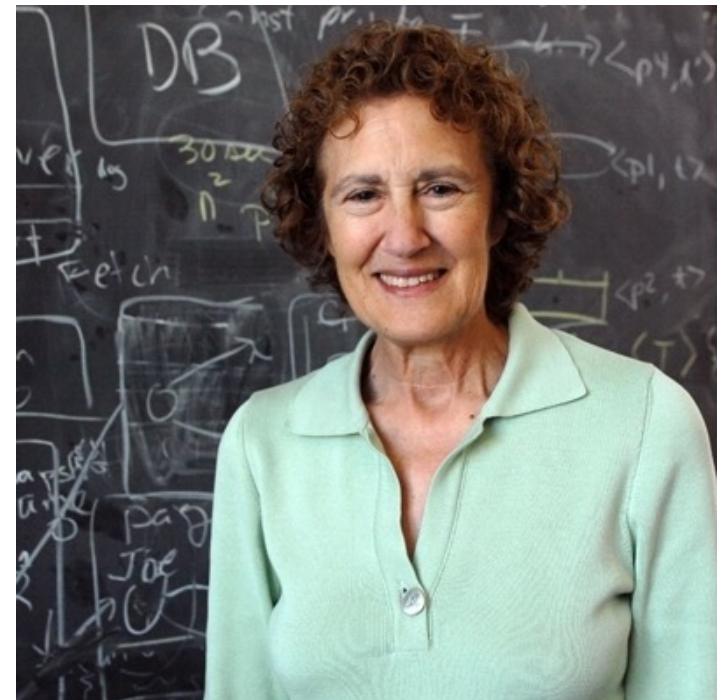


# Principe de substitution de Liskov

L'héritage est-il la bonne solution?

OUI SI les instances d'une classe peuvent être *remplacées par des instances de ses sous-classes sans altérer le programme.*

SINON il faut revoir la conception

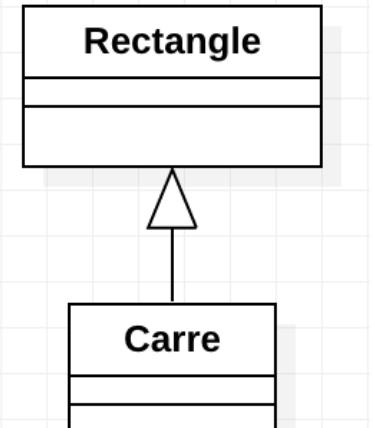


**Barbara LISKOV**

B. Liskov et J. Wing (dans “Family Values: A Behavioral Notion of Subtyping” (1993))

# Exemple 1 : Un carré est-il un rectangle?

```
public class Rectangle {  
    private int largeur;  
    private int longueur;  
    public Rectangle(int largeur, int longueur)  
    {  
        this.largeur = largeur;  
        this.longueur = longueur;  
    }  
    public int getLargeur() {  
        return largeur;  
    }  
    public void setLargeur(int largeur) {  
        this.largeur = largeur;  
    }  
    public int getLongueur() {  
        return longueur;  
    }  
    public void setLongueur(int longueur) {  
        this.longueur = longueur;  
    }  
    public int surface() {  
        return this.largeur*this.longueur;  
    } }
```



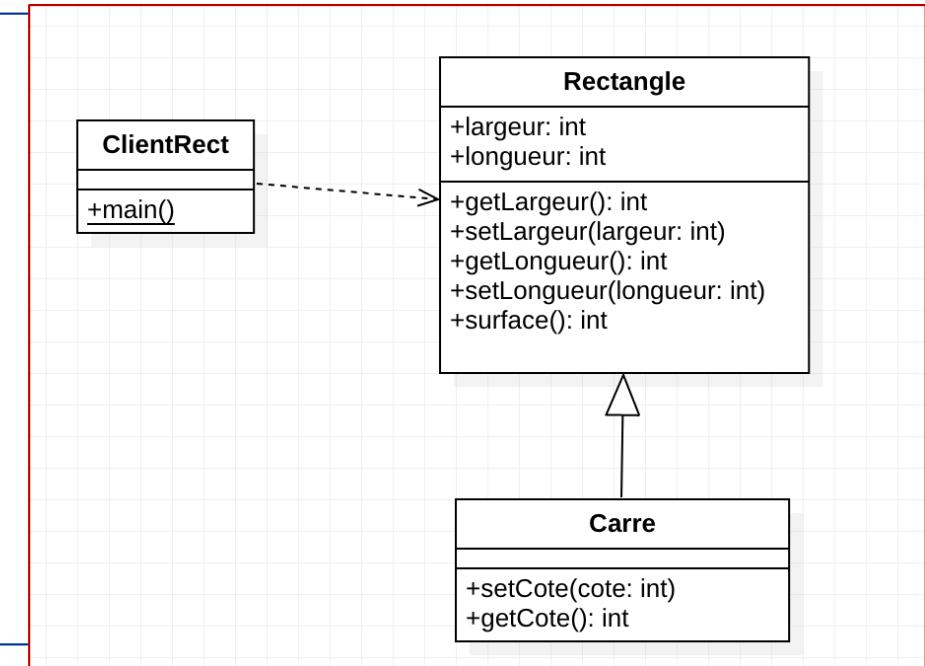
```
public class ClientRect {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(5,6);  
        r.setLargeur(10);  
        r.setLongueur(20);  
        System.out.println(r.surface());  
        //Affiche 200  
    } }
```

*Si un carré est un rectangle alors le programme doit fonctionner correctement si r est un carré!*



# Exemple 1 : Un carré est-il un rectangle?

```
public class Carre extends Rectangle{  
    public Carre(int cote) {  
        super(cote, cote);  
    }  
    public void setCote(int cote) {  
        this.setLargeur(cote);  
        this.setLongueur(cote);  
    }  
    public int getCote() {  
        return this.getLargeur();  
    }  
}
```



Le programme affiche bien 200 mais il y a un problème: Un carré ayant pour largeur 10 et longueur 20 est-il vraiment un carré??



```
public class ClientRect {  
    public static void main(String[] args)  
    {  
        Rectangle r = new Carre(5);  
        r.setLargeur(10);  
        r.setLongueur(20);  
        System.out.println(r.surface());  
        //doit afficher 200  
    }  
}
```

Test de substitution

# Exemple 1 : Un carré est-il un rectangle?

```
public class Carre extends Rectangle{  
    public Carre(int cote) {  
        super(cote, cote);  
    }  
    public void setCote(int cote) {  
        this.setLargeur(cote);  
        this.setLongueur(cote);  
    }  
    public int getCote() {  
        return this.getLargeur();  
    }  
    @Override  
    public void setLargeur(int largeur) {  
        super.setLargeur(largeur);  
        this.setLongueur(largeur);  
    }  
    @Override  
    public void setLongueur(int longueur) {  
        super.setLongueur(longueur);  
        this.setLargeur(longueur);  
    }  
}
```

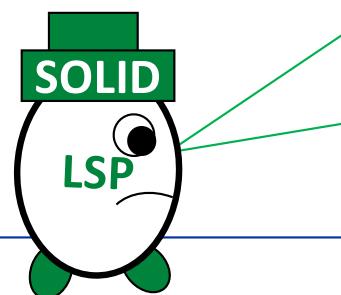
```
public class ClientRect {  
    public static void main(String[] args)  
    {  
        Rectangle r = new Carre(5);  
        r.setLargeur(10);  
        r.setLongueur(20);  
        System.out.println(r.surface());  
        //doit afficher 200  
    } }  
Test de substitution
```

ECHEC: Affiche 400!

Pas d'accord du tout !!

Le carré reste un carré mais le comportement du programme client a changé!

L'héritage n'est pas valide !!.



# Exemple 1 : Un carré est-il un rectangle?

```
public class Rectangle {  
    ...  
    public void setLargeur(int largeur){  
        this.largeur = largeur;  
        //Contrat (postcondition): largeur  
        modifiée et longueur non modifiée}...  
}
```

Exemple de violation de LSP

```
public class Carré extends Rectangle{  
    @Override  
    public void setLargeur(int largeur) {  
        super.setLargeur(largeur);  
        super.setLongueur(largeur);  
    } //Contrat violé: longueur aussi a été modifiée  
}
```



*La redéfinition n'est pas valide car elle viole le contrat initial! Elle viole donc LSP.*

Il faut revoir la conception

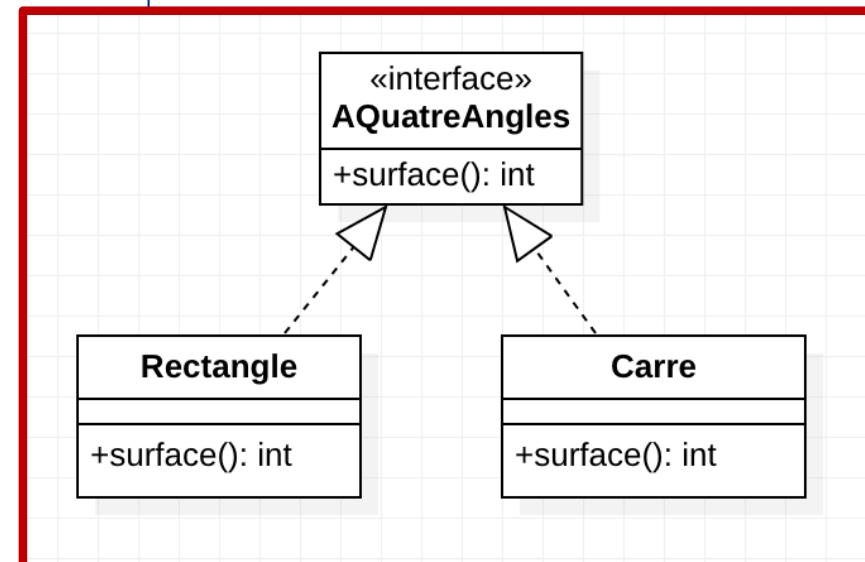
# Exemple 1 : NON : Carre ne doit pas être une sous-classe de Rectangle

## Solution 1: Utiliser une interface

```
public interface AQuatreAngles {  
    public int surface();}
```

```
public class Rectangle implements AQuatreAngles{  
    private int largeur;  
    private int longueur;  
    public int surface() {  
        return this.largeur*this.longueur;  
    } ...}
```

```
public class Carre implements AQuatreAngles{  
    int cote;  
    public Carre (int cote) {  
        this.cote=cote;}  
    public void setCote(int cote) {  
        this.cote=cote;}  
    public int getCote() {  
        return this.cote;}  
    public int surface() {  
        return this.cote*this.cote;  
    }}
```



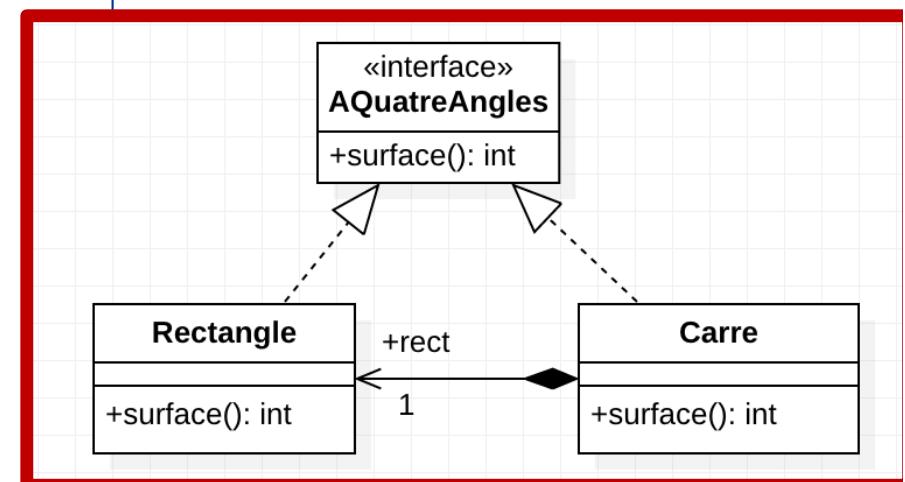
# Exemple 1 : NON : Carre ne doit pas être une sous-classe de Rectangle

## Solution 2: Interface + Composition

```
public interface AQuatreAngles {  
    public int surface();}
```

```
public class Rectangle implements AQuatreAngles{  
    private int largeur;  
    private int longueur;  
    public int surface() {  
        return this.largeur*this.longueur;  
    } ...}
```

```
public class Carre implements AQuatreAngles{  
    Rectangle rect;  
    public Carre (int cote) {  
        this.rect=new Rectangle(cote,cote);}  
    public void setCote(int cote) {  
        this.rect.setLargeur(cote);  
        this.rect.setLongueur(cote);}  
    public int getCote() {  
        return this.rect.getLargeur();}  
    public int surface() {  
        return this.rect.surface();}}
```



# Exemple 2 : Un pingouin est-il un oiseau?

```
public class Oiseau {  
    public void vole() {...  
}}
```

Exemple de violation de LSP

```
public class Perroquet extends Oiseau {  
    public void parle() {...  
}}
```

```
public class ClientOiseau {  
    public static void main(String[] args) {  
        Oiseau r = new Pingouin();  
        r.vole();  
        //doit voler  
    } }
```

Test de substitution

```
public class Pingouin extends Oiseau {  
    public void vole() {  
        //erreur: ne sait pas voler  
    } }
```

ECHEC: Affiche Erreur

*NON Je ne suis pas d'accord  
Votre conception ne signifie pas qu'un Pingouin ne peut pas voler.*

*Elle signifie: « Un pingouin peut voler mais s'il essaye cela déclenche une erreur ».*

*DONC un pingouin n'EST PAS un oiseau.*



Il faut revoir la conception



# Principe de substitution de Liskov

- Une méthode **redéfinie** ne doit pas :
  - Lancer des exceptions ou déclencher des erreurs non prévues dans la méthode de la classe mère
    - Préconditions plus fortes: conditions plus restrictives  
(ex=  $x < 10$  : la méthode déclenche une erreur si  $x < 10$  alors que dans la classe mère la méthode fonctionnait pour tout  $x$  entier )
  - Ne rien faire...
  - Avoir des post-conditions plus faibles:
    - Post-conditions de la méthode de la classe mère qui ne sont plus vérifiées

## En résumé

L'héritage n'est pas toujours la bonne solution: les redéfinitions qui violent LCP indiquent une erreur de conception

# Exercice : LSP1



```
class Stylo extends Crayon {  
    private final int limiteEncre;  
    private int quantiteEncre;  
    public Stylo(int quantiteEncre) {  
        super();  
        this.quantiteEncre = quantiteEncre;  
        this.limiteEncre = quantiteEncre / 2;  
    }  
    public void ecrire(String texte) {  
        if (!estPlein())  
            throw new RuntimeException("Presque plus  
d'encre!");  
        ecrire0(texte);  
        diminuerEncre();  
    }  
    public void remplir() {  
        quantiteEncre = limiteEncre * 2;  
    }  
    private boolean estPlein() {  
        return quantiteEncre > limiteEncre;  
    }  
    private void ecrire0(String texte) {  
        System.out.println(texte);  
    }  
    private void diminuerEncre() {  
        quantiteEncre--;  
    }  
}
```

Définissez une classe de test afin de démontrer que la classe Stylo viole le principe LSP et expliquez où se situe le problème.

```
abstract class OutilEcriture {  
    public abstract void ecrire(String texte);  
}
```

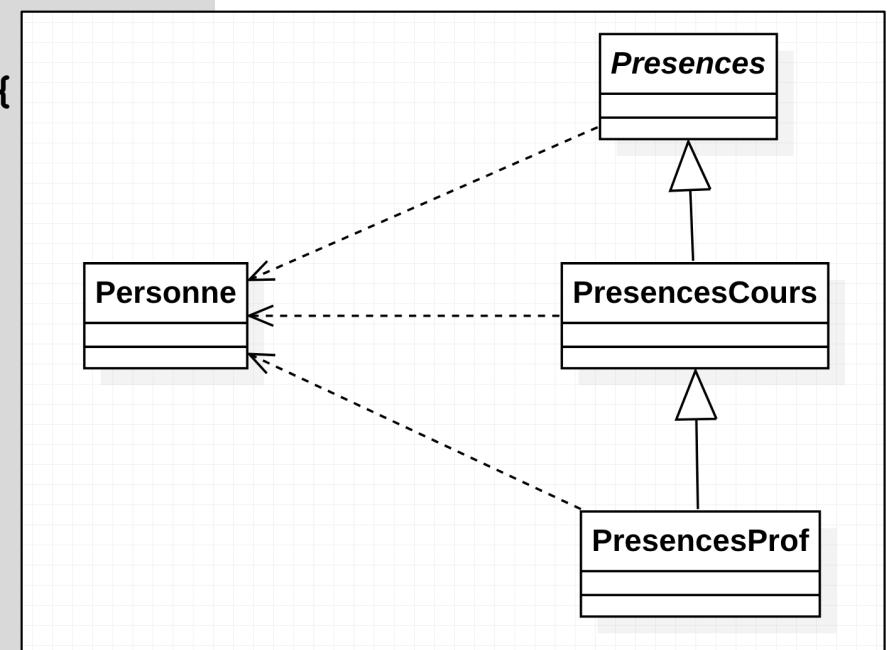
```
public class Crayon  
    extends OutilEcriture {  
    public void ecrire(String texte)  
    {  
        System.out.println(texte);  
    }  
}
```

# Exercice : LSP2 (1/3)



```
public class Personne {  
    private boolean estProf;  
    private String nom;  
  
    public Personne(boolean estProf, String name) {  
        super();  
        this.estProf = estProf;  
        this.nom = name;  
    }  
    public void sePresente( String msg ) {  
        System.out.println(msg);  
    }  
    public boolean estProf() {  
        return estProf;  
    }  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

On considère les classes du diagramme UML ci-dessous.



# Exercice : LSP2 (2/3)



```
public abstract class Presences {  
    public abstract void arriveEnCours(Personne p) ;  
}
```

file  
d'attente

```
public class PresencesCours extends Presences {  
    private int numArrivee = 0;  
    private HashMap<String, Integer> registre = new HashMap<String, Integer>();  
  
    public void arriveEnCours(Personne p) {  
        signalerArrivee(p);  
        enregistreArrivee(p);  
        //PostCondition: p est dans le registre  
    }  
  
    protected void enregistreArrivee(Personne p) {  
        numArrivee++;  
        registre.put(p.getNom(), numArrivee);  
    }  
  
    protected void signalerArrivee(Personne p) {  
        p.sePresente(p.getNom()+" entre en cours");  
    }  
  
    public String toString() {  
        return "Presences en cours [registre=" + registre + "]";  
    }  
}
```

# Exercice : LSP2 (3/3)

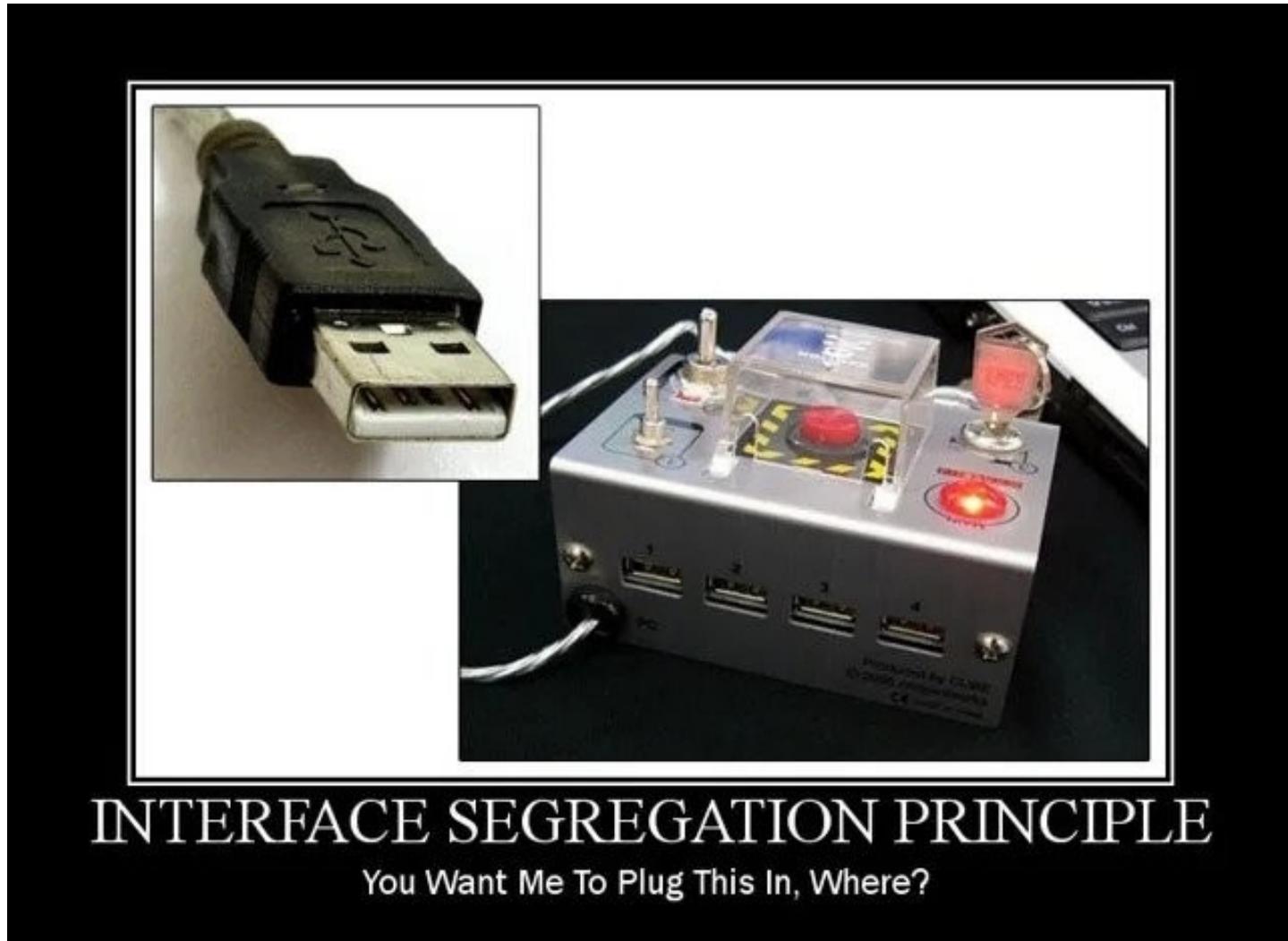


Définissez une classe de test afin de démontrer que la classe **PresencesProf** viole le principe LSP et expliquez où se situe le problème.

```
public class PresencesProf extends PresencesCours
{
    public void arriveEnCours(Personne p) {
        if (p.estProf()) {
            signalerArrivee(p);
            enregistreArrivee(p);
            presenterCours(p);
        }
    }
    private void presenterCours(Personne p) {
        p.sePresente(p.getNom() + " est enseignant
dans ce cours");
    }
}
```



# Principe de ségrégation de l'Interface

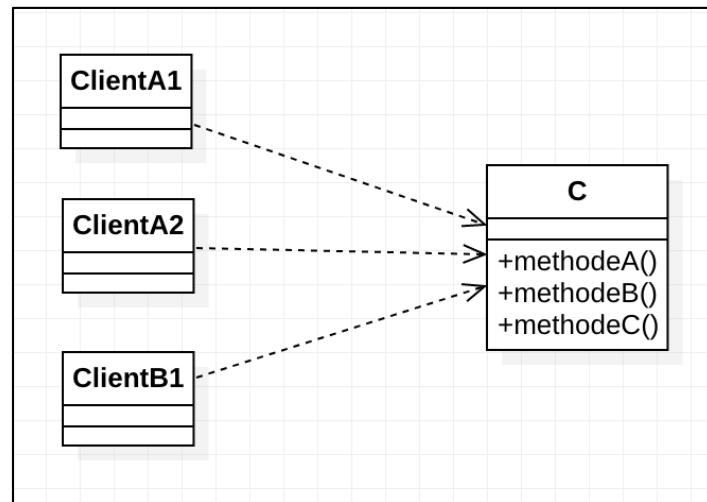


*Vous voulez que je connecte cette prise. Mais où?*



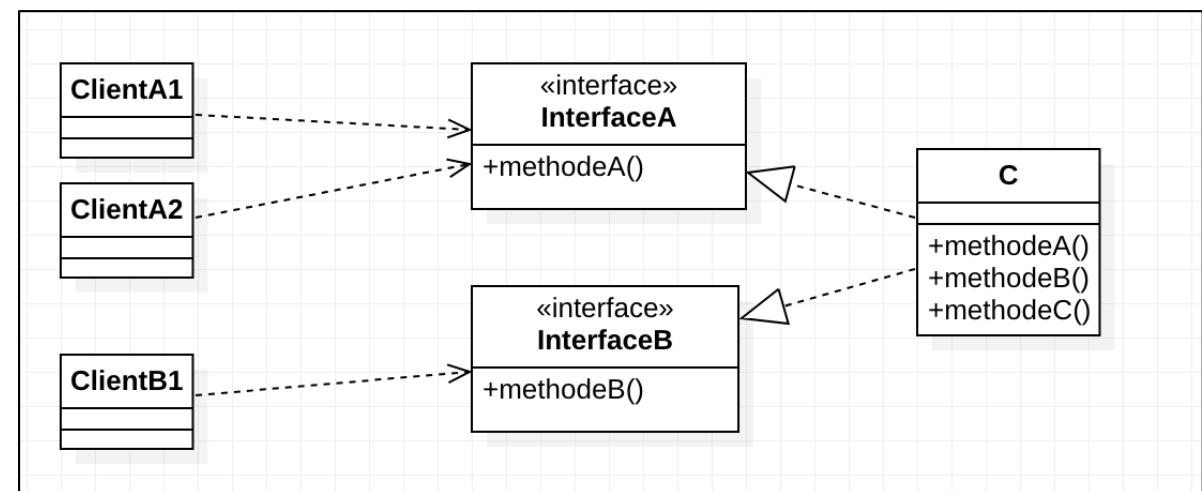
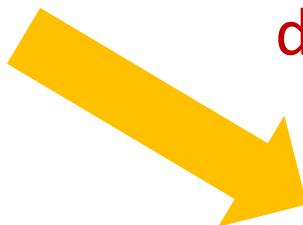
# Principe de Ségrégation de l'Interface

- Créer des interfaces spécifiques pour chaque type de client.



- ClientA1 et ClientA2 n'utilisent que methodeA()
- ClientB1 n'utilise que methodeB()

Un client doit juste dépendre de ce dont il a besoin.





# Principe de Ségrégation de l'Interface Intérêt?

- Limiter l'impact des modifications
  - Ex: Si la méthodeC() est modifiée cela n'aura aucun impact sur les clients clientA1, clientA2 et clientB1
  - Eviter les codes « vides » et les exceptions en cas d'implémentation d'une interface dont toutes les méthodes ne sont pas utilisées
- Attention à l'excès!
  - Ne pas créer systématiquement une interface par méthode!!

*Ne pas oublier  
LSP*



# Exercice : ISP



Le code suivant ne respecte pas ISP.  
Expliquez pourquoi et proposez une autre solution d'organisation.

```
Public public interface Animal {  
    void voler();  
    void courir();  
    void aboyer();}  
  
public class Oiseau implements Animal {  
    public void aboyer() { /* ne rien faire */ }  
    public void courir() { ... }  
    public void voler() { ... }  
}  
  
public class Chat implements Animal {  
    public void voler() {  
        throw new Exception("propriété non définie"); }  
    public void aboyer() {  
        throw new Exception("propriété non définie"); }  
    public void courir() { ... }  
}  
  
public class Chien implements Animal {  
    public void voler() { /* ne rien faire */ }  
    public void aboyer() { ... }  
    public void courir() { ... }  
}
```



# Principe d’Inversion des Dépendances



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

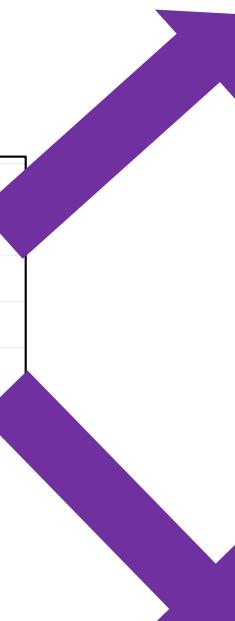
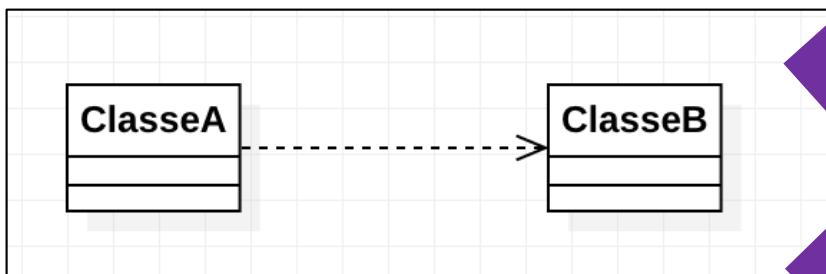
*Souderiez-vous une lampe directement sur  
le câblage électrique dans le mur?*



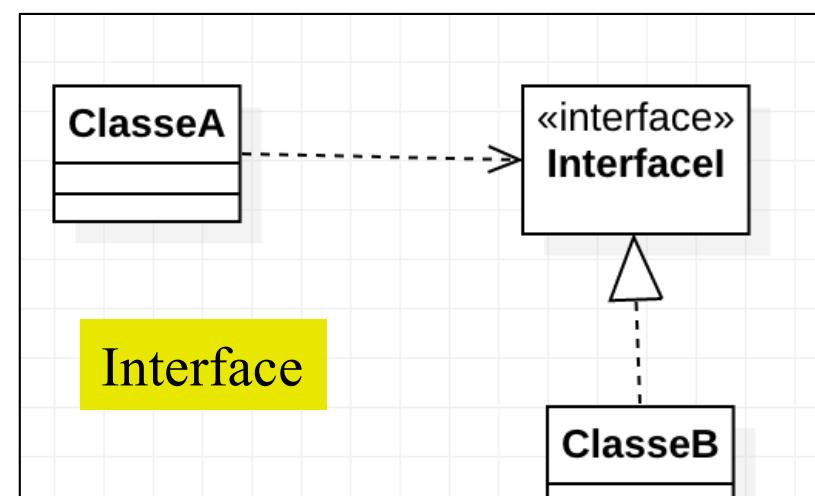
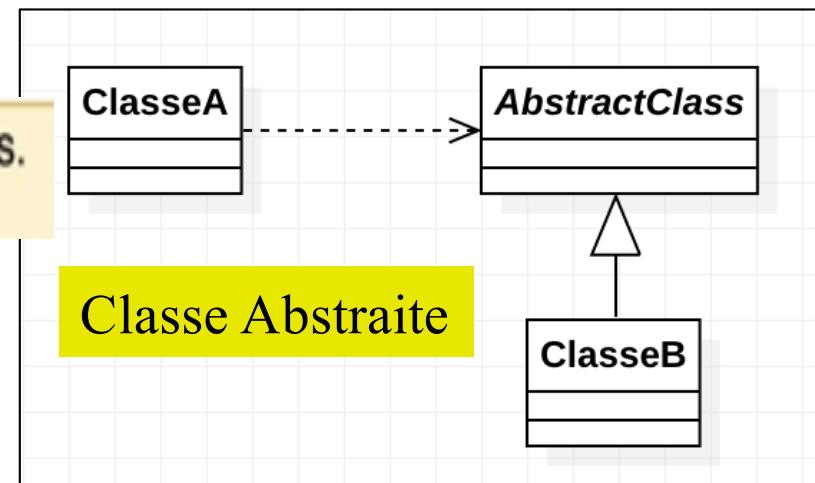
# Principe d'Inversion des Dépendances

- Dépendre des **abstractions**, ne pas dépendre des classes concrètes.

High-level modules should not depend on low-level modules.  
Both should depend on abstractions.



Classes abstraites, Interfaces

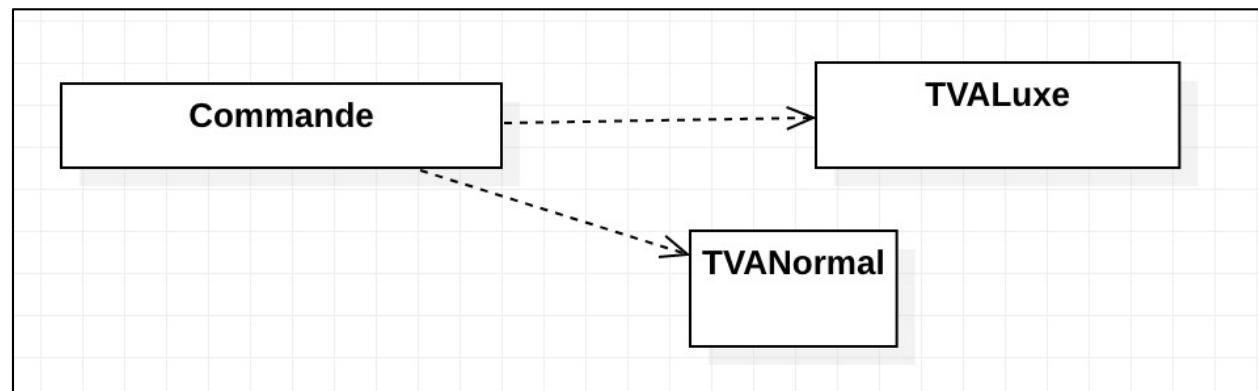




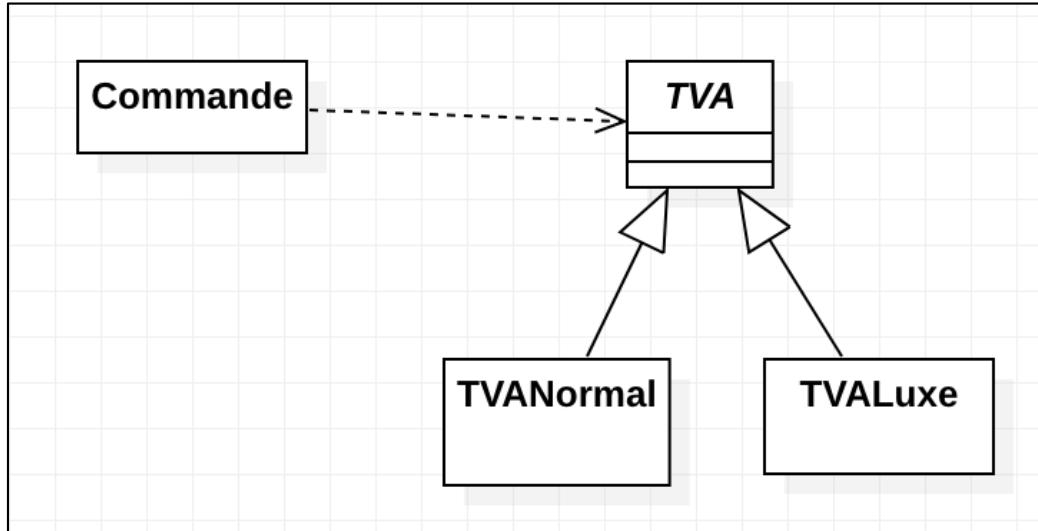
# Exemple violation DIP

```
public class Commande {  
  
    private double taux;  
  
    public Commande(String typeTVA) {  
        if (typeTVA.equals("Luxe")) {  
            TVALuxe tva=new TVALuxe();  
            this.taux=tva.getTaux();  
        }  
        else  
            if (typeTVA.equals("Normal")) {  
                TVANormal tva=new TVANormal();  
                this.taux=tva.getTaux();  
            }  
    }  
}
```

- Commande dépend de classes concrètes
- Problèmes Evolutivité
  - Nouveau taux TVA?
  - Commande ne vérifie pas non plus OCP



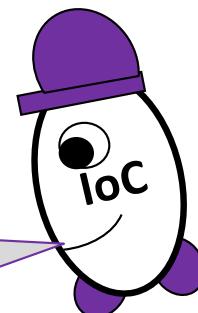
# Exemple violation DIP : Solution



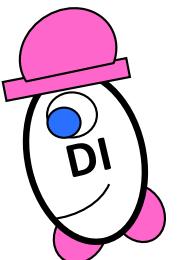
- Commande ne dépend plus des classes concrètes
- Mais concrètement, comment faire pour instancier tva?

```
public class Commande {  
    private double taux;  
    public Commande() {  
        TVA tva=new ???;  
        this.taux=tva.getTaux();  
    }  
}
```

Avec DI,  
nous avons  
une  
solution!



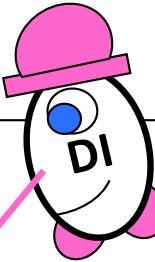
« Inversion Of  
Control »



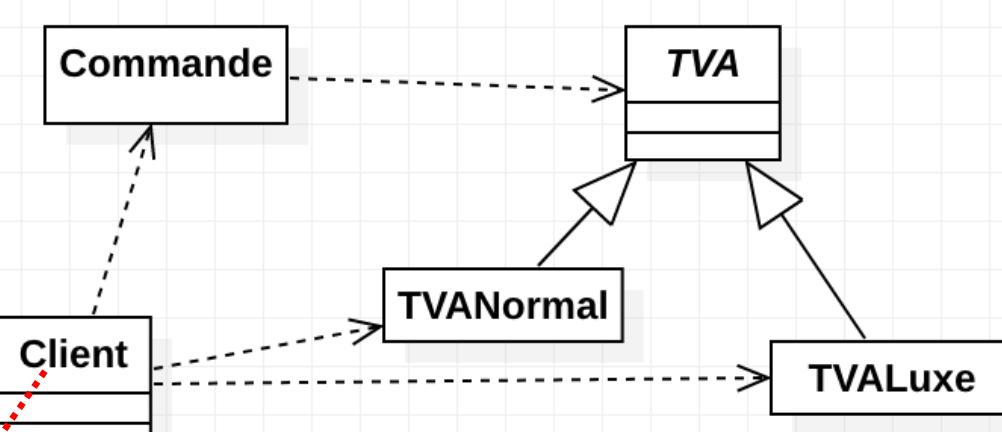
« Dependency  
Injection »

# Exemple violation DIP : Solution

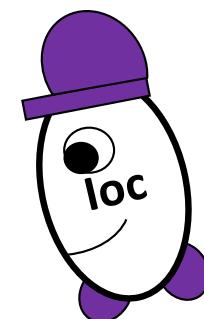
```
public class Commande {  
  
    private double taux;  
    public Commande(TVA tva) {  
        this.taux=tva.getTaux();  
    }  
}
```



- *DI Injection de dépendances*  
Les dépendances vers TVALuxe et TVANormal sont injectées par le paramètre du constructeur
- *IoC Inversion du contrôle:*  
Commande perd le contrôle de la création d'instances de TVALuxe et de TVANormal



classe TestCommande





# Exercice : DIP1



Récupérer sur l'ENT le code des classes Commande, TVA, TVALuxe, TVANormal et TestCommande.

- Modifiez le code de la classe Commande conformément à la diapo précédente.
- Réécrivez le code de la classe TestCommande.
- Expliquez où se situe **l'injection de dépendance**

# DIP, IoC et DI: Notions liées

## DIP : Le Principe

Une classe ne doit pas dépendre de classes concrètes mais d'abstractions

## IoC : La solution *(Inversion Of Control)*

## DI : Une technique *(Dependency Injection)*

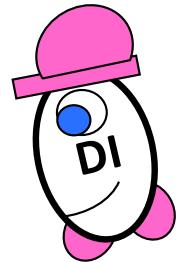


# DIP et OCP

- DIP et OCP sont étroitement liés
- L'application du principe DIP garantit la vérification d'OCP:
  - Une classe qui ne dépend que d'abstractions vérifie par définition OCP

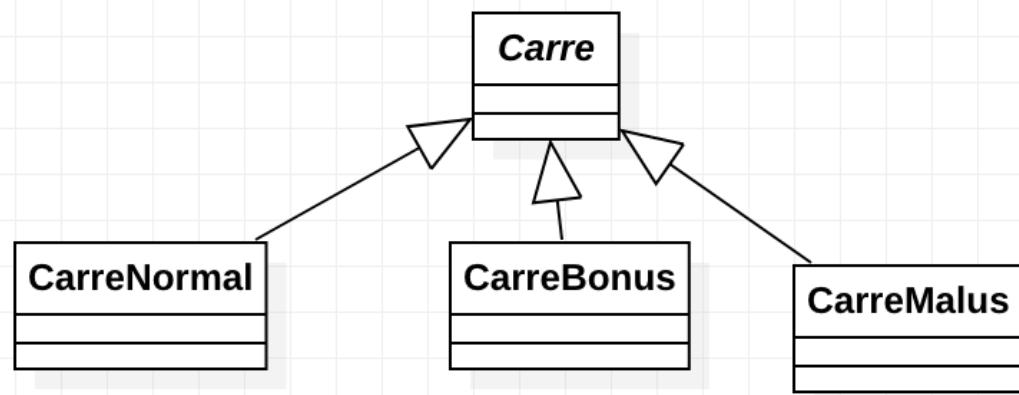


# Exercice : DIP2



La classe Jeu ne respecte ni OCP ni DIP.  
Expliquez pourquoi et proposez une  
solution pour y remédier.

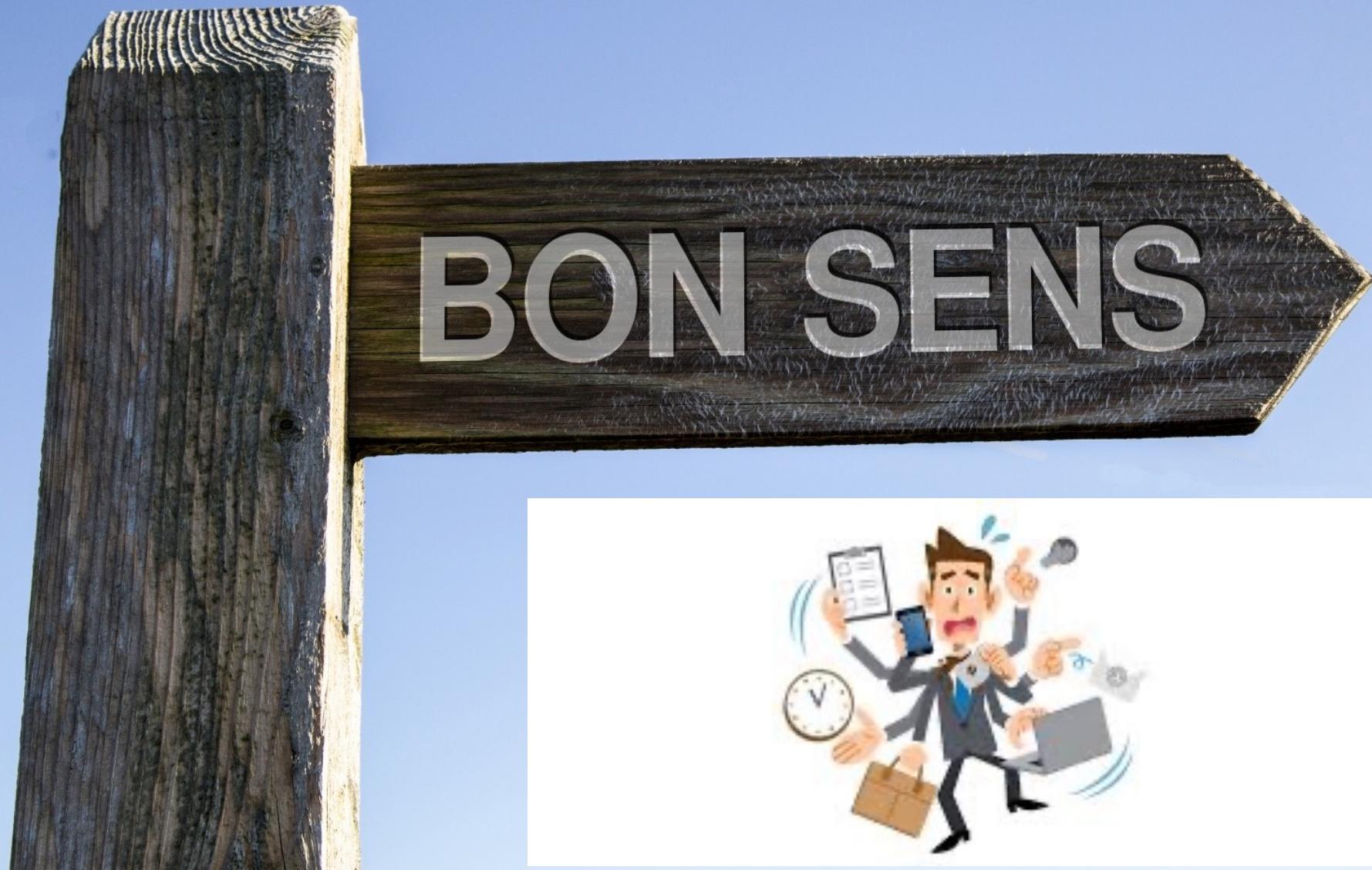
```
public class Jeu {  
    public int getPoints(Carre s) {  
        int nbPoints=0;  
        if (s instanceof CarreNormal)  
            nbPoints=20;  
        else  
            if (s instanceof CarreBonus)  
                nbPoints=50;  
            else  
                if (s instanceof  
                    CarreMalus)  
                    nbPoints=-40;  
        return nbPoints;  
    }  
}
```



# Pour terminer, deux derniers principes à ne pas oublier!



Les principes de conception sont un guide.  
Ils doivent être utilisés sans excès.



# Quelques références pour en savoir plus

1. "The Influence of SOLID Principles in the Quality of Object-Oriented Software: A Systematic Literature Review" par Luiz Fernando de Oliveira, et al., publié dans le journal IEEE Access en 2020.
2. "The Adoption of SOLID Principles in Open-Source Software: An Empirical Study" par Dara Ghasemi et al., publié dans le journal Empirical Software Engineering en 2021.