

# Adapter et Façade

Par

- COSSA Azário
- DESCOUTURES Cathy
- GENDRY Marine
- GERBEAUD Evan

# Sommaire

- Introduction
- Exemple
- Pattern Adapter et Façade
- Principes SOLID
- Limites
- Architecture Hexagonale
- Exemple 2
- Live coding
- Conclusion
- Bibliographie
- QCM

# Introduction

## Catégories de Design Patterns

- **Créationnels** : Gèrent la création d'objets.
- **Structurels** : Définissent la composition des classes et objets.
- **Comportementaux** : Régissent les interactions entre objets.

## Objectifs

- **Vocabulaire commun** entre développeurs
- **Réutilisabilité** et **maintenabilité** accrues

## Focus : Patterns Structurels

- **Adapter** : Rend compatibles des interfaces différentes.
- **Facade** : Simplifie l'accès à des systèmes complexes.



# Exemple

## **Contexte**

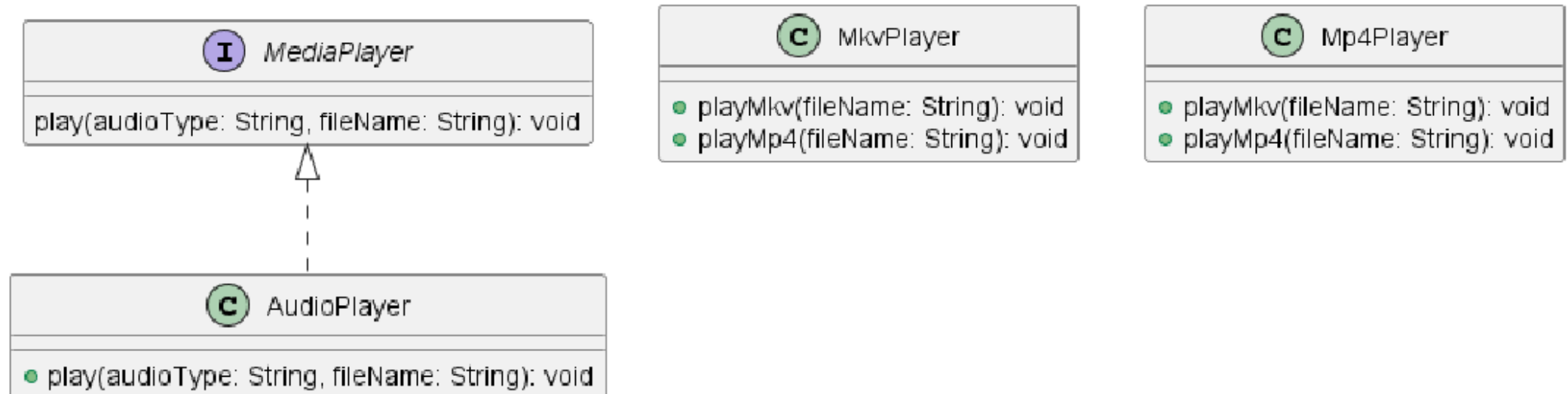
Dans notre application multimédia, nous avons un lecteur audio capable de lire des fichiers uniquement au format MP3, mais on souhaite qu'il puisse aussi lire d'autres formats comme le MP4 ou le MKV, sans modifier l'implémentation de la classe existante.

## **Besoin :**

L'objectif est de permettre à l'application multimédia de lire aussi les fichiers MP4 et MKV, ou autre format que nous pouvons vouloir lire éventuellement.

# Example

---



# Solution

Le **Pattern Adapter** permet d'ajouter de nouveaux formats audios à un lecteur sans modifier le code existant.

## Problèmes identifiés

1. **Incompatibilité d'interfaces** : Le MediaPlayer ne peut fonctionner qu'avec une AudioPlayer, rendant le code inflexible.
2. **Couplage fort** : Le MediaPlayer est couplé à la classe AudioPlayer, ce qui ne permet pas d'utiliser d'autres types de lecteur.

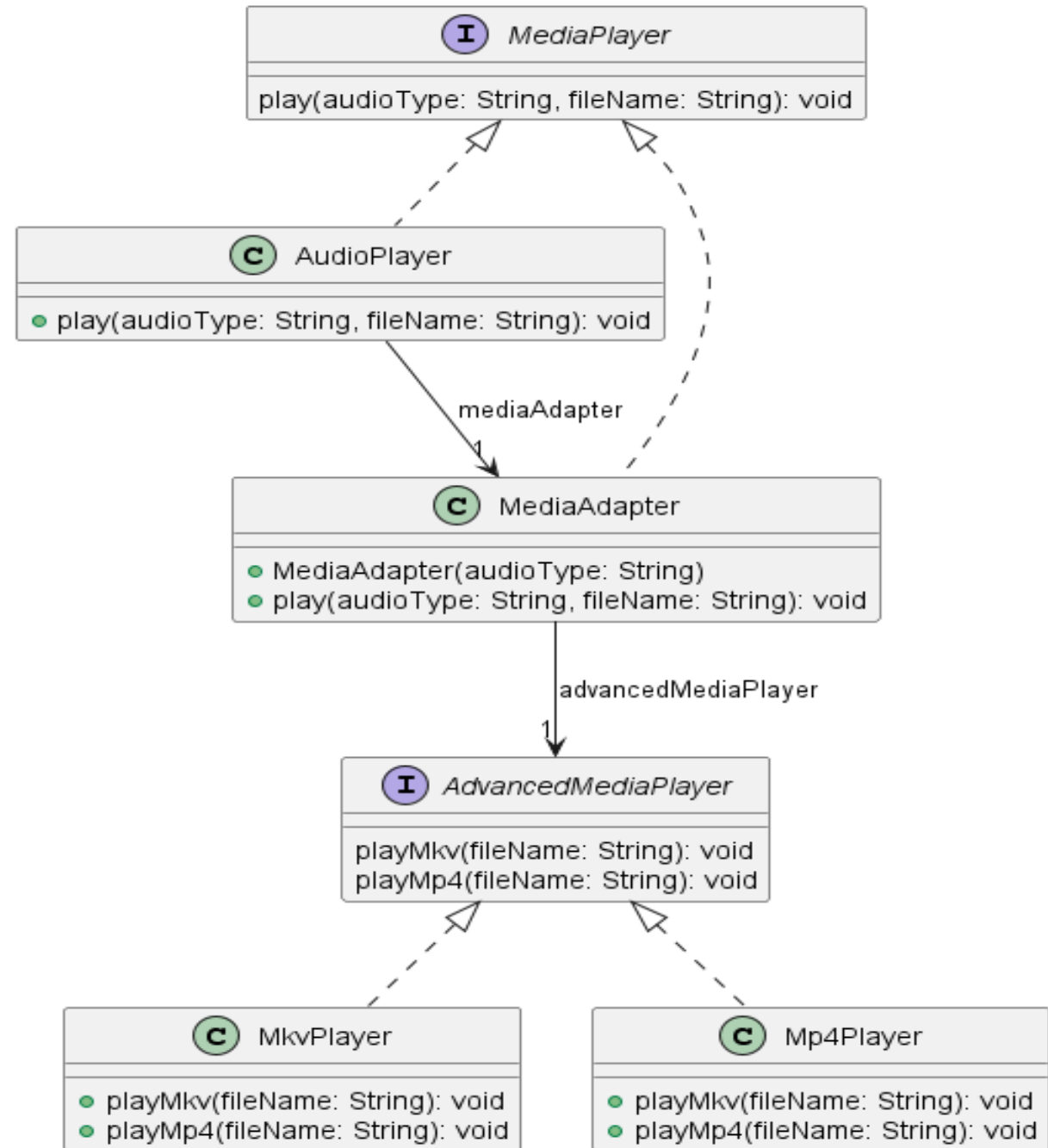
## Solution : Utilisation du pattern Adapter

Pour résoudre ce problème, nous allons :

1. Créer l'interface MediaPlayer : Définir la méthode play qui sera utilisée à la fois par AudioPlayer et par MediaAdapter pour lire les fichiers média.
2. Implémenter AudioPlayer : Cette classe lit les fichiers MP3 nativement et utilise MediaAdapter pour gérer d'autres formats, comme le MP4.
3. Créer l'adaptateur MediaAdapter : Implémente MediaPlayer pour lire les fichiers MP4, le rendant ainsi compatible avec AudioPlayer.

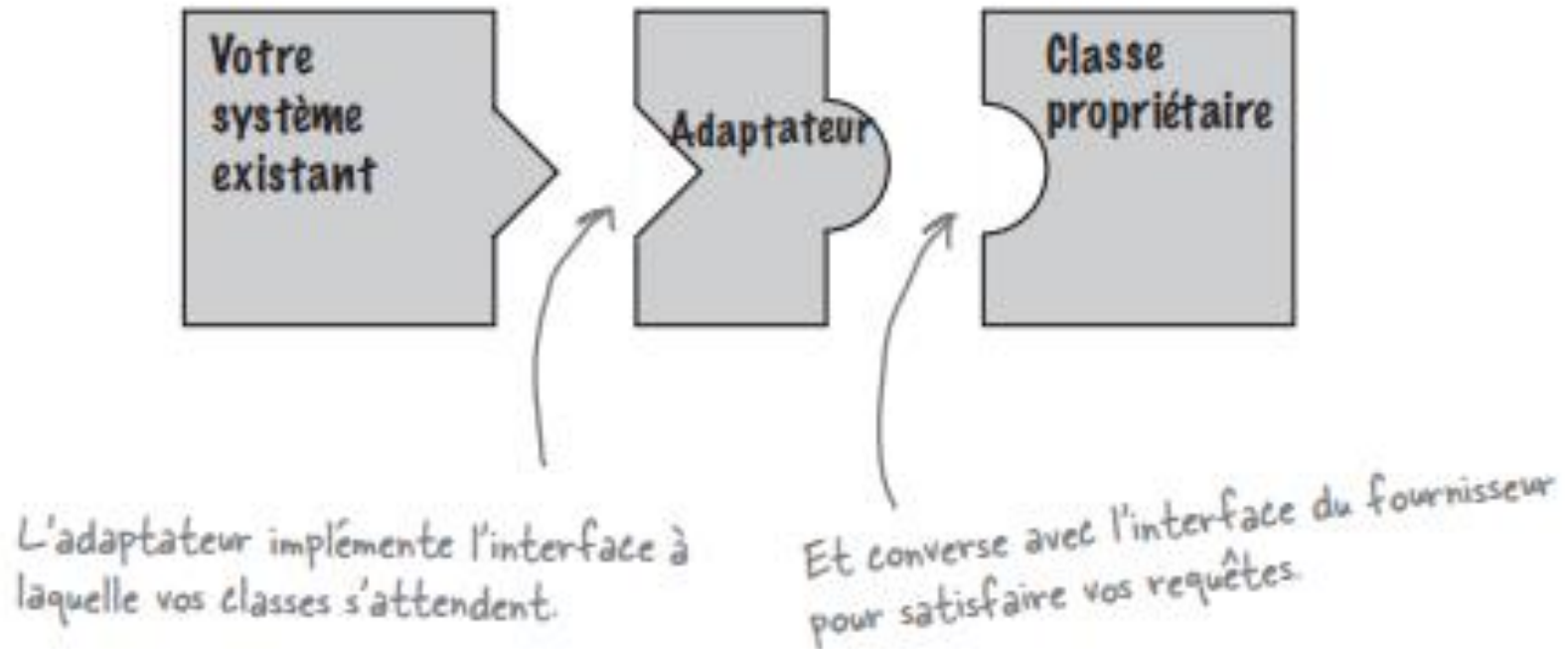


# Solution



# Fonctionnement du Pattern

Le pattern **Adapter** est utilisé pour permettre à des classes ayant des interfaces incompatibles de travailler ensemble. Il agit comme un "pont" entre deux interfaces incompatibles.







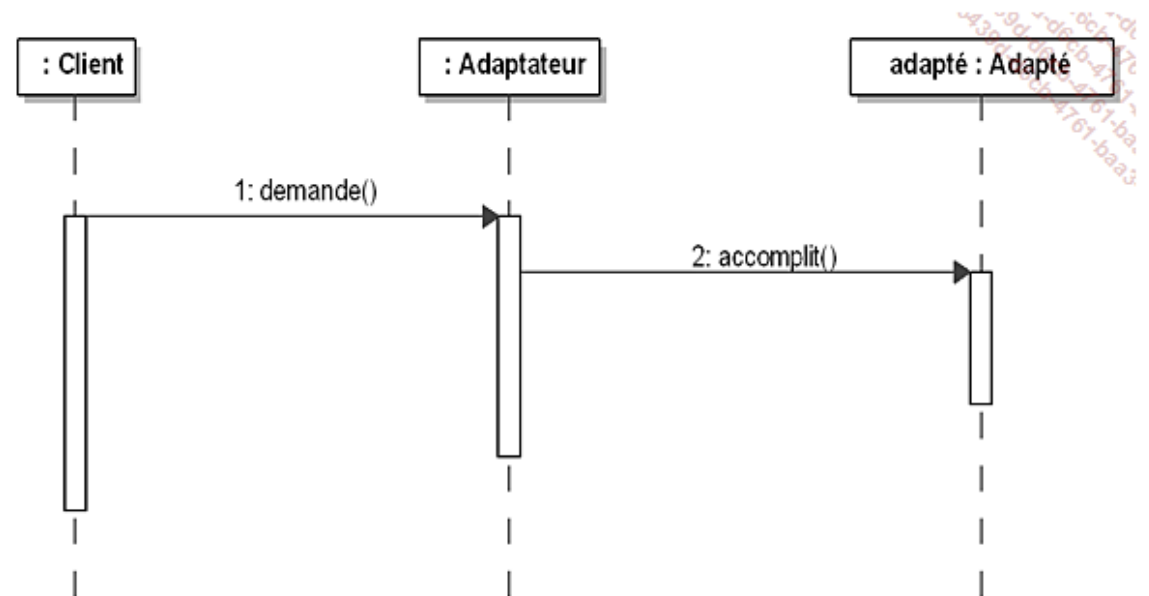
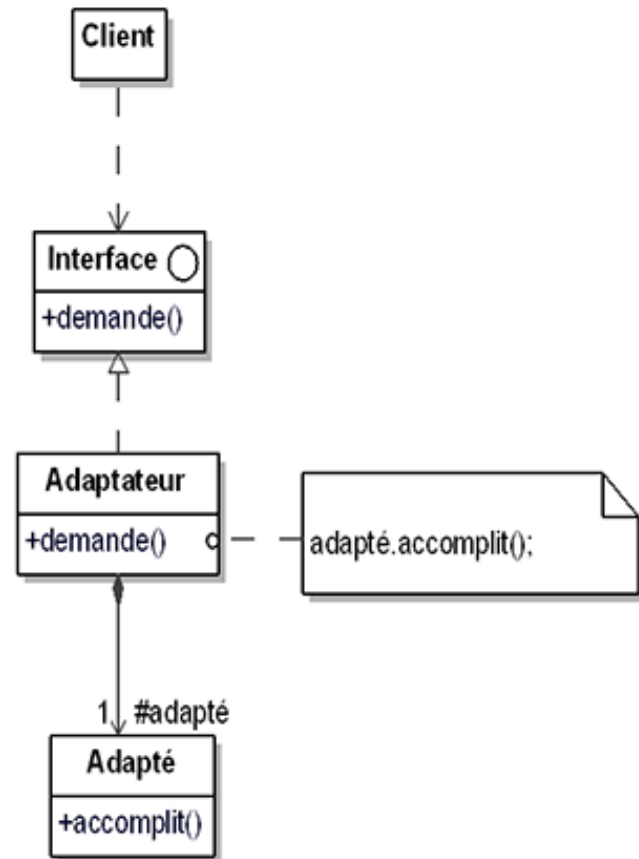
Ce pattern permet d'intégrer une classe avec une interface différente dans un système existant.

Il existe 2 types de structures:

- Adaptateur d'objet
- Adaptateur de classe (marche qu'avec de l'héritage multiple comme en c++, pas notre cas ici)



# Diagramme de classe et séquence du pattern Adapter selon le GOF



# Principes SOLID

---

## Adapter

Respecté :

- Single Responsibility Principle (SRP)
  - L'Adaptateur a une seule responsabilité : adapter l'interface source à l'interface cible.
- Open/Closed Principle (OCP)
  - On peut ajouter de nouveaux Adaptateurs pour d'autres interfaces sans modifier les classes existantes.
- Liskov Substitution Principle (LSP)
  - Cela dépend de l'Adaptateur. S'il ne respecte pas correctement l'interface cible, des problèmes peuvent apparaître.
- Interface Segregation Principle (ISP)
  - L'Adaptateur permet d'exposer seulement ce qui est nécessaire pour le client, sans surcharger les interfaces d'origine.
- Dependency Inversion Principle (DIP)
  - Le client dépend d'une abstraction (l'interface cible) et non d'une implémentation concrète.

# Limites du pattern Adapter

- **Complexité** : L'ajout d'adaptateurs pour chaque interface incompatible peut alourdir le système.
- **Performance** : Les appels supplémentaires de l'adaptateur peuvent légèrement impacter les performances.
- **Rigidité** : Des changements dans les interfaces nécessitent des modifications des adaptateurs, réduisant la flexibilité.

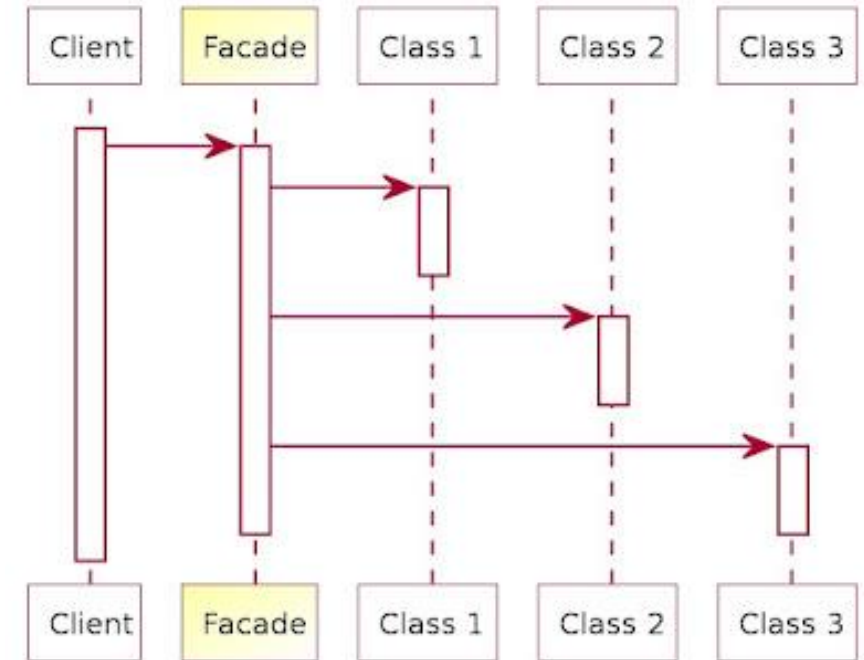
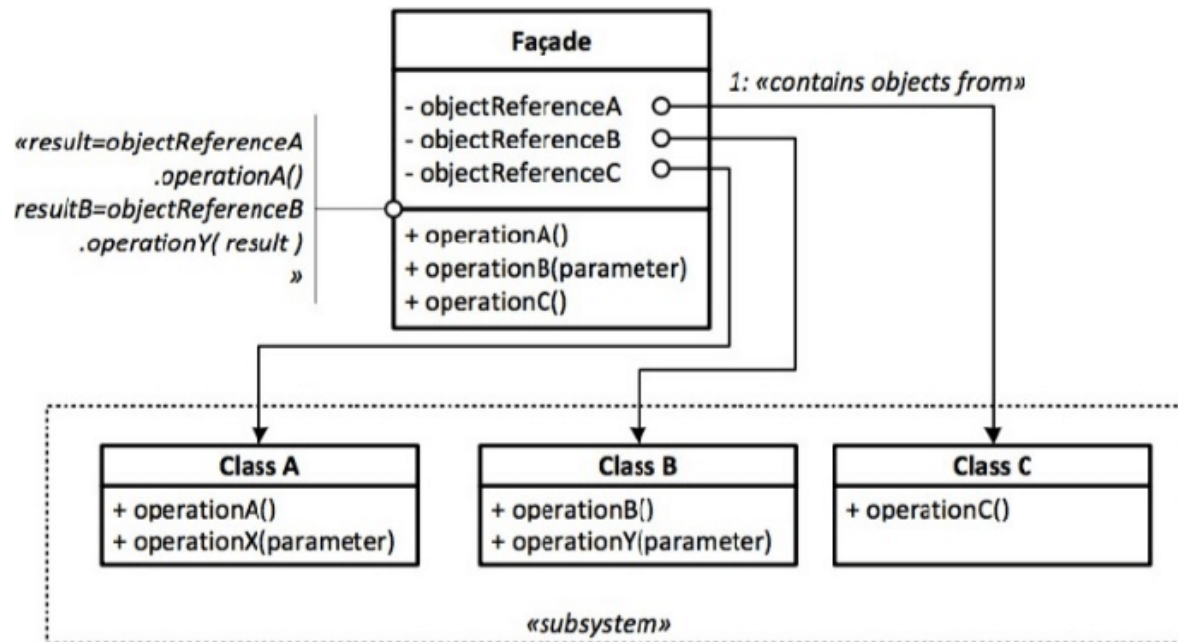


# Pattern Façade

La Façade est un patron structurel qui simplifie l'accès à une librairie, un framework ou un ensemble complexe de classes.

La Façade crée une nouvelle interface pour un sous-système, tandis que l'Adaptateur adapte une interface existante pour un seul objet.

# Diagramme classe et séquence du pattern Façade



Sample sequence diagram

# Principes SOLID

---

## Façade

Respecté :

- Single Responsibility Principle (SRP)
  - Se limite à fournir un point d'accès simplifié vers le sous-système.
- Dependency Inversion Principle (DIP)
  - La Façade peut respecter le DIP si elle dépend d'abstractions du sous-système plutôt que d'implémentations concrètes.

Non respecté :

- Open/Closed Principle (OCP)
  - Peut être difficile à étendre sans modifier la classe elle-même, surtout si de nouvelles fonctionnalités sont ajoutées au sous-système.
- Liskov Substitution Principle (LSP)
  - La Façade est une classe unique qui encapsule plusieurs composants sans hériter d'autres classes, ce qui diffère du principe du pattern.
- Interface Segregation Principle (ISP)
  - La Façade peut enfreindre le principe ISP, car elle expose une interface globale qui peut contenir des méthodes inutiles pour certains clients.



# Limites du pattern Façade

- **Couplage indirect** : La Façade crée une dépendance unique, nécessitant des adaptations en cas de changement du sous-système.
- **Complexité masquée** : La Façade complique le débogage et l'accès aux fonctionnalités avancées.
- **Risque de surcharge** : Une Façade trop complexe perd son objectif de simplification.



# Lien entre Décorateur et Façade

- Adapter : Change l'interface d'un objet pour le rendre compatible avec un autre contexte.
- Décorateur : Ajoute des fonctionnalités sans modifier l'interface.
- Façade : Simplifie l'accès à un système complexe.

Adapter et Décorateur modifient le comportement des objets, tandis que Façade en simplifie l'utilisation.

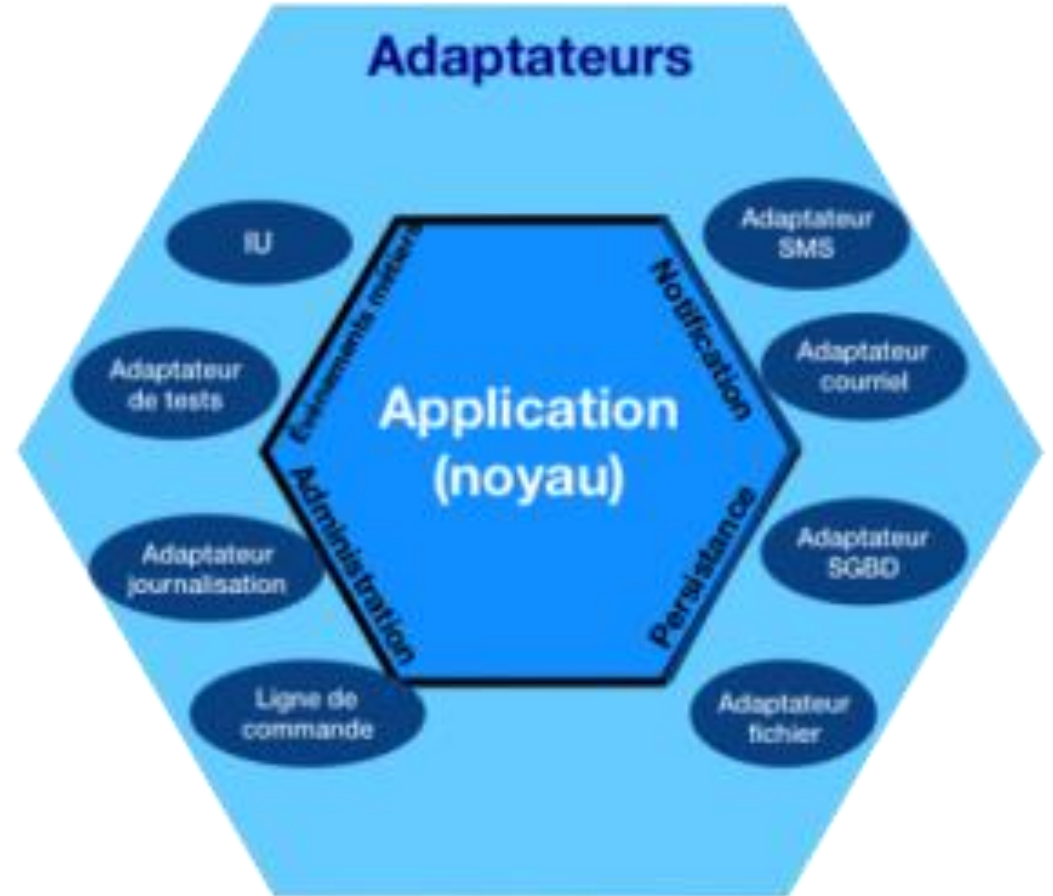
# Architecture Hexagonale

---

- **Flexibilité** pour changer de technologie sans affecter la logique métier.
- **Testabilité** accrue grâce à l'isolation de la logique métier.
- **Extensibilité** pour connecter de nouveaux services externes.


**Ports:** Interface, ce que l'application doit faire

**Adapters:** adapter les éléments externes au format du domaine





## Exemple 2 Adapter



Notre plateforme de livraison en ligne de nourriture utilise un format XML pour collecter les données des différents restaurants.

Pour améliorer l'application, nous avons besoin d'intégrer une bibliothèque qui utilise le format JSON.

Afin de l'utiliser sans la modifier et éviter tout dysfonctionnement, nous devons mettre en place un adaptateur.

```
public interface InterfaceApplicationRestaurant {

    void afficherMenus(DonneesXML donneesXML);

    void afficherRecommandations(DonneesXML donneesXML);

}
```

```
public class BibliServiceUI {
    public void afficherMenus(DonneesJSON donnéeJSON) {
        //TODO menu avec données JSON
    }

    public void afficherRecommandations(DonneesJSON donnéeJSON) {
        // TODO recommandation avec données JSON
    }
}
```

```
public class ApplicationRestaurant implements InterfaceApplicationRestaurant {
    @Override
    public void afficherMenus(DonneesXML donneesXML) {
        System.out.println("Afficher les menus avec les données XML...");
    }

    @Override
    public void afficherRecommandations(DonneesXML donneesXML) {
        System.out.println("Afficher les recommandations avec les données XML...");
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        DonneesXML mesDonnees = new DonneesXML();

        // Ancienne UI
        InterfaceApplicationRestaurant appResto = new ApplicationRestaurant();
        appResto.afficherMenus(mesDonnees);
        appResto.afficherRecommandations(mesDonnees);

        System.out.println("=====");

        // Nouvelle UI
        InterfaceApplicationRestaurant adapter = new AdaptateurBibliServiceUI();
        adapter.afficherMenus(mesDonnees);
        adapter.afficherRecommandations(mesDonnees);

    }

}
```

```
public class AdaptateurBibliServiceUI implements InterfaceApplicationRestaurant {
    private final BibliServiceUI bibliServiceUI;

    public AdaptateurBibliServiceUI() {
        bibliServiceUI = new BibliServiceUI();
    }

    @Override
    public void afficherMenus(DonneesXML donneesXML) {
        DonneesJSON donneesJSON = conversionXMLenJSON(donneesXML);
        System.out.println("Affichage des Menus avec des données JSON...");
        bibliServiceUI.afficherMenus(donneesJSON);
    }

    @Override
    public void afficherRecommandations(DonneesXML donneesXML) {
        DonneesJSON donneesJSON = conversionXMLenJSON(donneesXML);
        System.out.println("Affichage des recommandation avec des données JSON...");
        bibliServiceUI.afficherRecommandations(donneesJSON);
    }

    private DonneesJSON conversionXMLenJSON(DonneesXML donneesXML) {
        System.out.println("Conversion des données XML en données JSON...");
        return new DonneesJSON();
    }
}
```

```
Afficher les menus avec les données XML...
Afficher les recommandations avec les données XML...
=====
Conversion des données XML en données JSON...
Affichage des Menus avec des données JSON...
Conversion des données XML en données JSON...
Affichage des recommandation avec des données JSON...
```



## Exemple 2

### Façade

On a un café et on veut gérer de manière plus simple les commandes de nos clients.

Dans notre café nous avons plusieurs rôles comme un caissier, un serveur ou encore un Barista

```
public class Caisier {
    public void encaisser(String commande) {
        System.out.println("Traitement du paiement pour la commande: " + commande);
    }
}
```

```
public class Barista {
    public void prepareCafe(String cafe) {
        System.out.println("Préparation du café: " + cafe);
    }
}
```

```
public class main {
    public static void main(String[] args) {
        FacadeDuCafe cafe = new FacadeDuCafe();
        cafe.completeLaCommande("Chocolat Chaud");
    }
}
```

```
Prise de commande: Chocolat Chaud
Préparation du café: Chocolat Chaud
Service du café: Chocolat Chaud
Traitement du paiement pour la commande: Chocolat Chaud
```

```
public class Serveur {

    public void prendreLaCommande(String commande) {
        System.out.println("Prise de commande: " + commande);
    }

    public void servirLeCafe(String cafe) {
        System.out.println("Service du café: " + cafe);
    }
}
```

```
public class FacadeDuCafe {
    private Serveur serveur;
    private Barista barista;
    private Caisier caisier;

    public FacadeDuCafe() {
        this.serveur = new Serveur();
        this.barista = new Barista();
        this.caisier = new Caisier();
    }

    public void completeLaCommande(String commande) {
        serveur.prendreLaCommande(commande);
        barista.prepareCafe(commande);
        serveur.servirLeCafe(commande);
        caisier.encaisser(commande);
    }
}
```



# Live Coding

<https://youtu.be/JyvRHXxQMh8?si=F9rRlsutF3N6uaHS>



# Live coding Pattern Adapter

## Contexte :

Notre application possède un système de paiement utilisant des Euros (€).

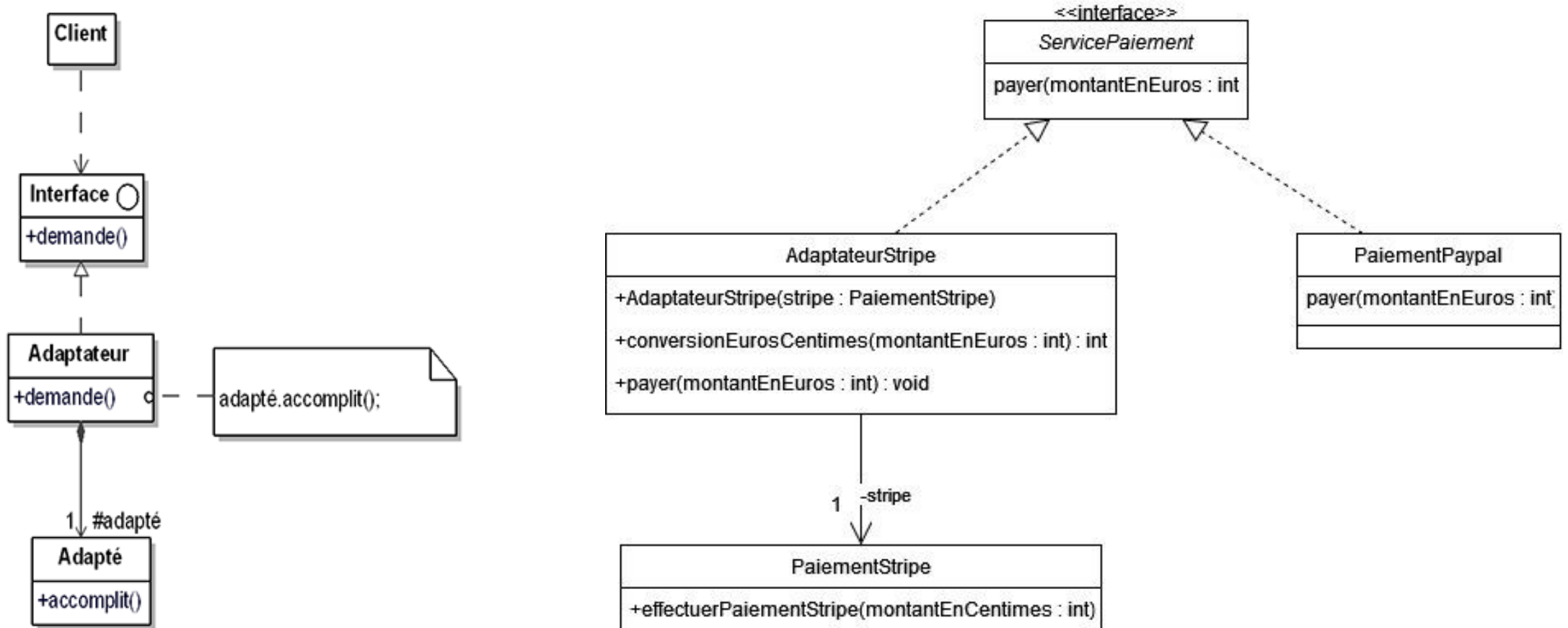
## Problème :

L'application n'accepte que PayPal (en Euros) et doit intégrer Stripe, qui utilise des centimes.

## Solution :

Utiliser un adaptateur pour convertir les centimes de Stripe en Euros, facilitant l'intégration tout en gardant la cohérence du système.

# Diagramme de classe Adapter





# Live coding Pattern Façade

## Contexte :

L'application de commerce en ligne gère les commandes via des sous-systèmes : inventaire, paiement et logistique.

## Problème :

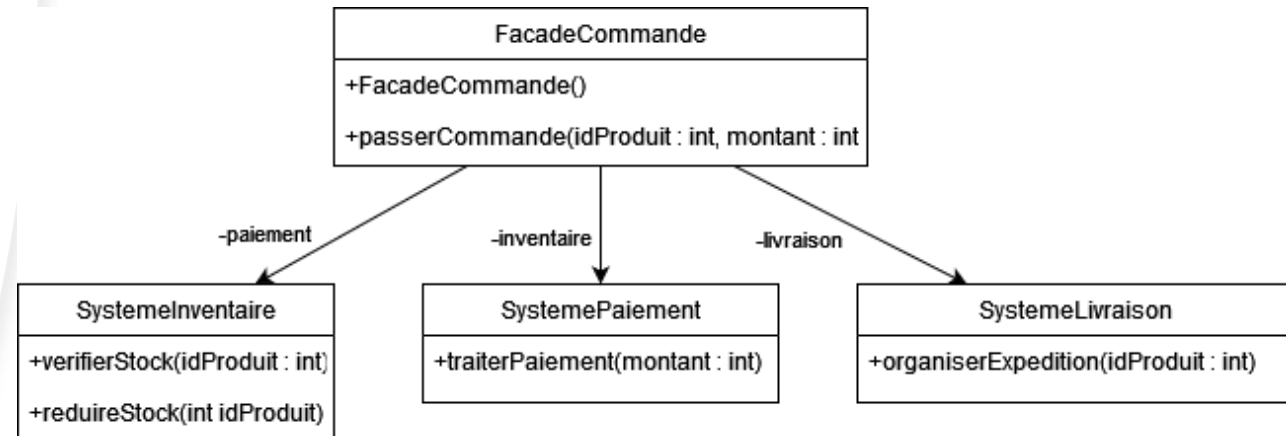
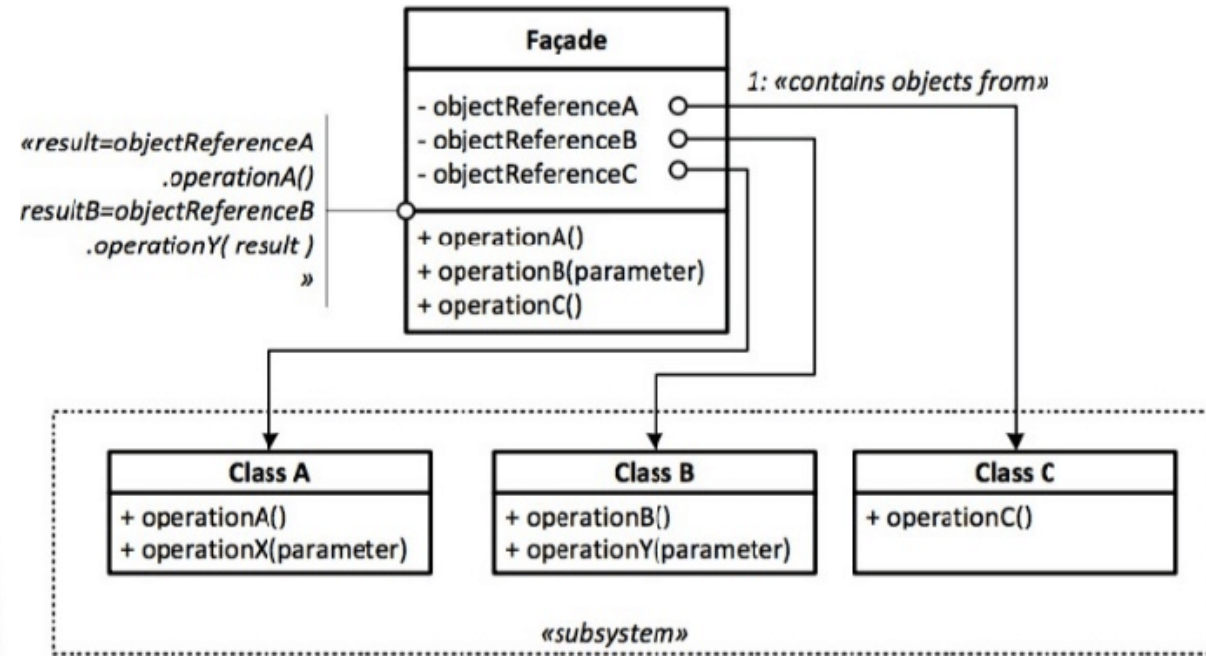
Le client doit gérer chaque sous-système séparément pour passer commande, ce qui complexifie le code et augmente le couplage.

## Solution :

Nous utiliserons le pattern Façade pour offrir un point d'entrée unique et une interface simplifiée, masquant la complexité des sous-systèmes.

# Diagramme de classe Façade

27



# Conclusion

Les patterns **Adapter** et **Façade** améliorent la flexibilité, la lisibilité et la maintenabilité du code en adaptant et en encapsulant les systèmes existants.



# Bibliographie

- [Site où y'a les infos Adaptateur](#)
- [Site où y'a les infos Façade](#)
- [Diapo avec tous les patterns ?](#)
- <https://fuhrmanator.github.io/log210-ndc-quarto/GRASP-GoF.html>
- <https://www.editions-eni.fr/livre/design-patterns-en-java-descriptions-et-solutions-illustrees-en-uml-2-et-java-5e-edition-les-23-modeles-de-conception-9782409037603/le-pattern-adapter>
- [https://perso-laris.univ-angers.fr/~delanoue/polytech/design\\_pattern/support/bates\\_design-patterns-tete-la-premiere.pdf](https://perso-laris.univ-angers.fr/~delanoue/polytech/design_pattern/support/bates_design-patterns-tete-la-premiere.pdf)
- <https://www.sfeir.dev/back/les-design-patterns-structurels-facade/>





QCM