

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

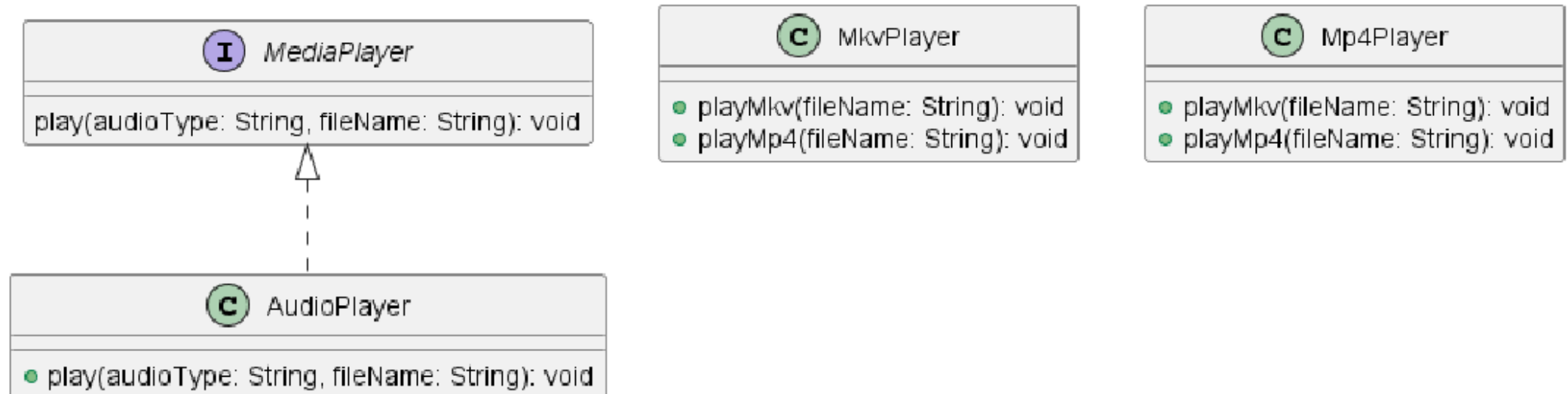
Contexte

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 de "brancher" de nouveaux formats de lecture audio à notre lecteur, tout en maintenant la compatibilité avec les formats existants, sans modifier le code déjà écrit.

Problèmes identifiés

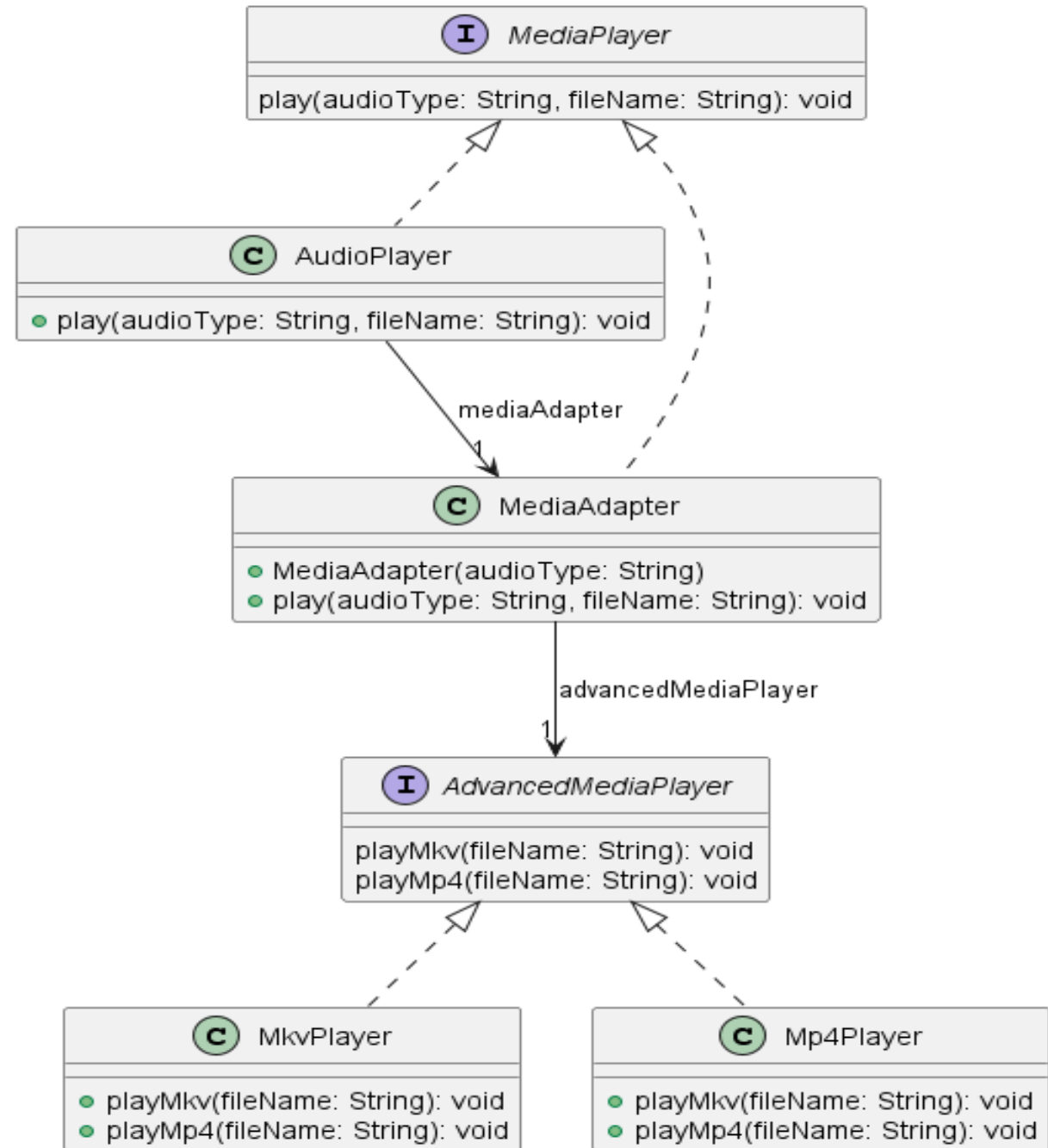
1. **Incompatibilité d'interfaces** : Le MediaPlayer ne peut fonctionner qu'avec une AudioPlayer, rendant le code inflexible.
2. **Couplage fort** : Le MediaPlayer est couplé à la classe AudioPlayer, ce qui ne permet pas d'utiliser d'autres types de lecteur.

Solution : Utilisation du pattern Adapter

Pour résoudre ce problème, nous allons :

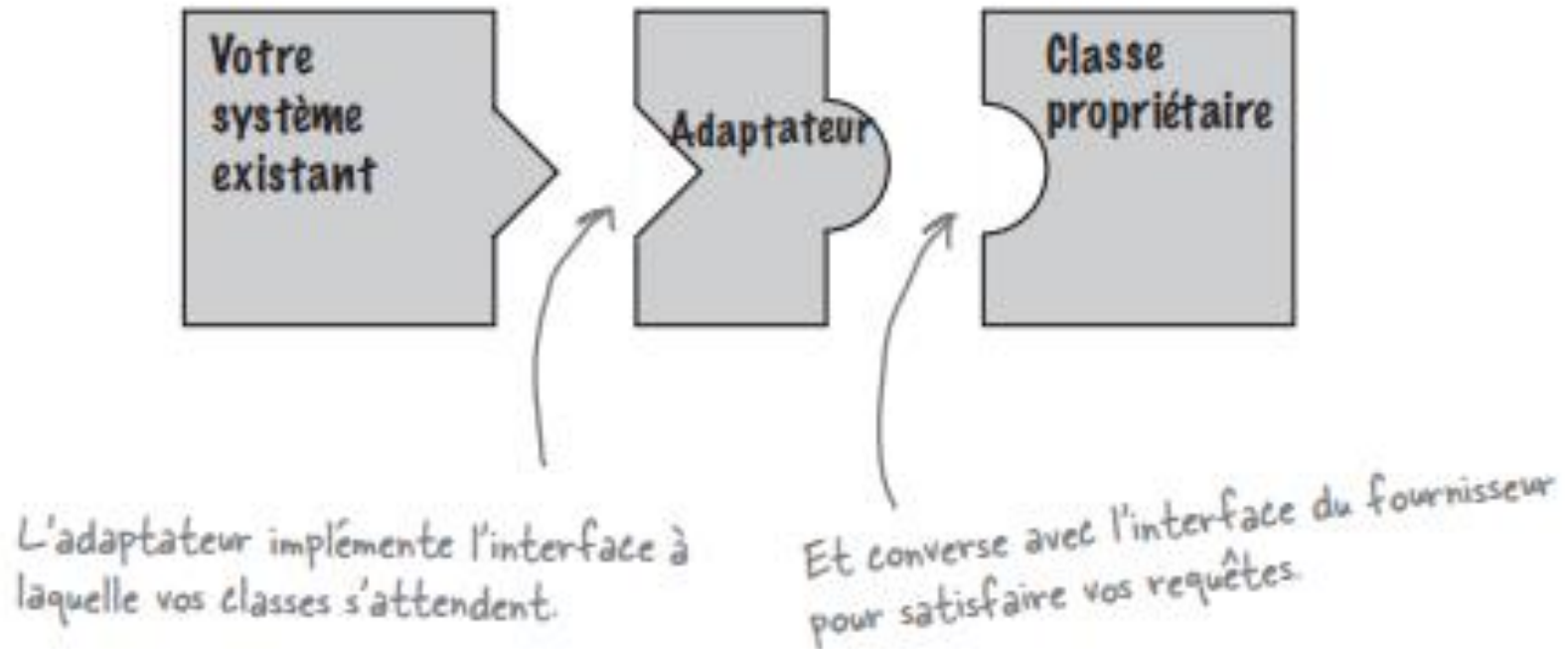
1. Créer l'interface MediaPlayer : Définir la méthode play qui sera utilisée à la fois par AudioPlayer et par MediaAdapter pour lire les fichiers média.
2. Implémenter AudioPlayer : Cette classe lit les fichiers MP3 nativement et utilise MediaAdapter pour gérer d'autres formats, comme le MP4.
3. Créer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



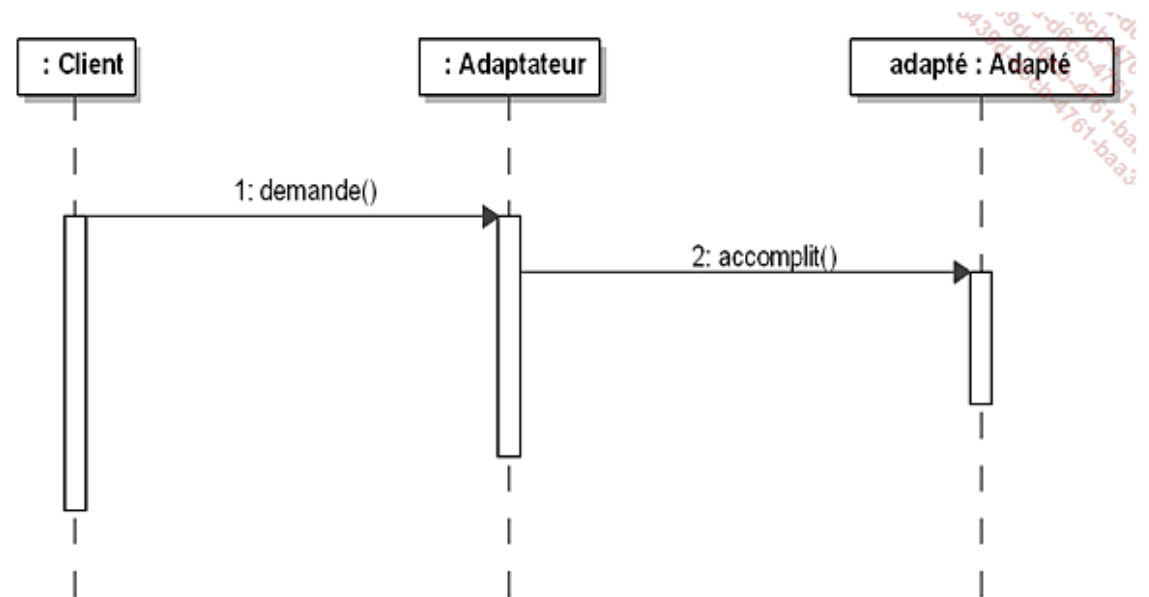
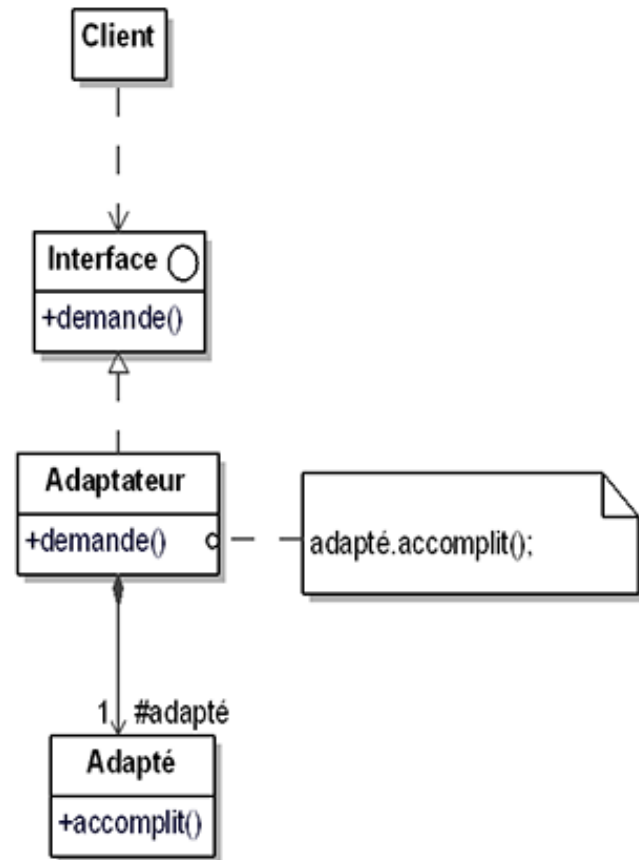


Ce pattern est souvent employé lorsque l'on veut intégrer une nouvelle classe dans un système existant, mais que cette nouvelle classe n'a pas la même interface que les autres composants.

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'Adapter a une seule responsabilité : adapter l'interface source à l'interface cible.
- Open/Closed Principle (OCP)
 - On peut ajouter de nouveaux Adapters pour d'autres interfaces sans modifier les classes existantes.
- Liskov Substitution Principle (LSP)
 - Dépend de la manière dont l'Adapter est conçu. Si l'Adapter ne respecte pas l'interface cible de façon correcte, cela peut causer des problèmes.
- Interface Segregation Principle (ISP)
 - L'Adapter 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 additionnelle** : La création d'un adaptateur pour chaque interface incompatible peut ajouter de la complexité au système.
- **Performance** : Les appels de méthode supplémentaires introduits par l'adaptateur peuvent légèrement affecter la performance, bien que ce soit souvent négligeable.
- **Rigidité en cas de changement** : Si l'interface cible ou celle de la classe adaptée change, l'adaptateur doit être modifié pour continuer à fonctionner, ce qui peut nuire à la flexibilité du système.

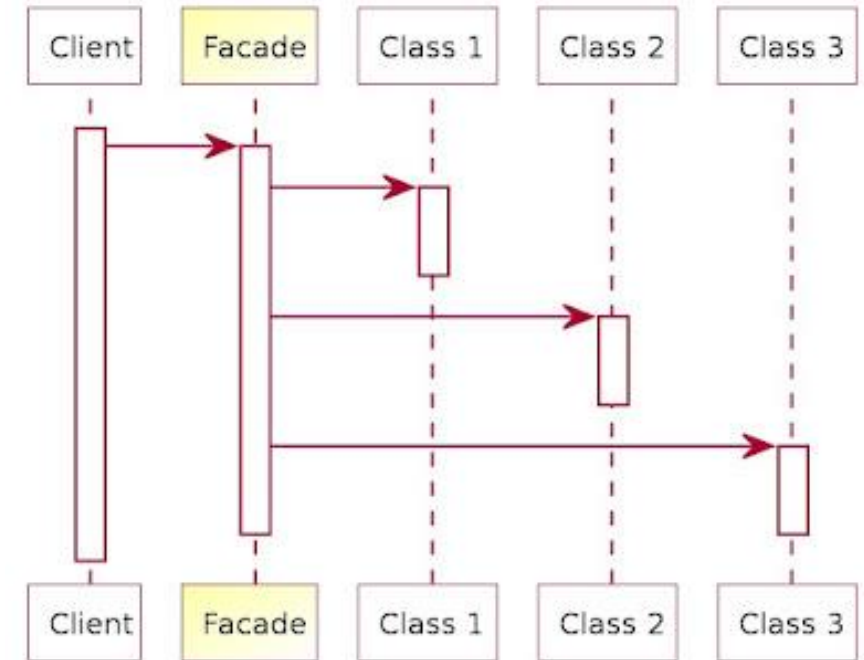
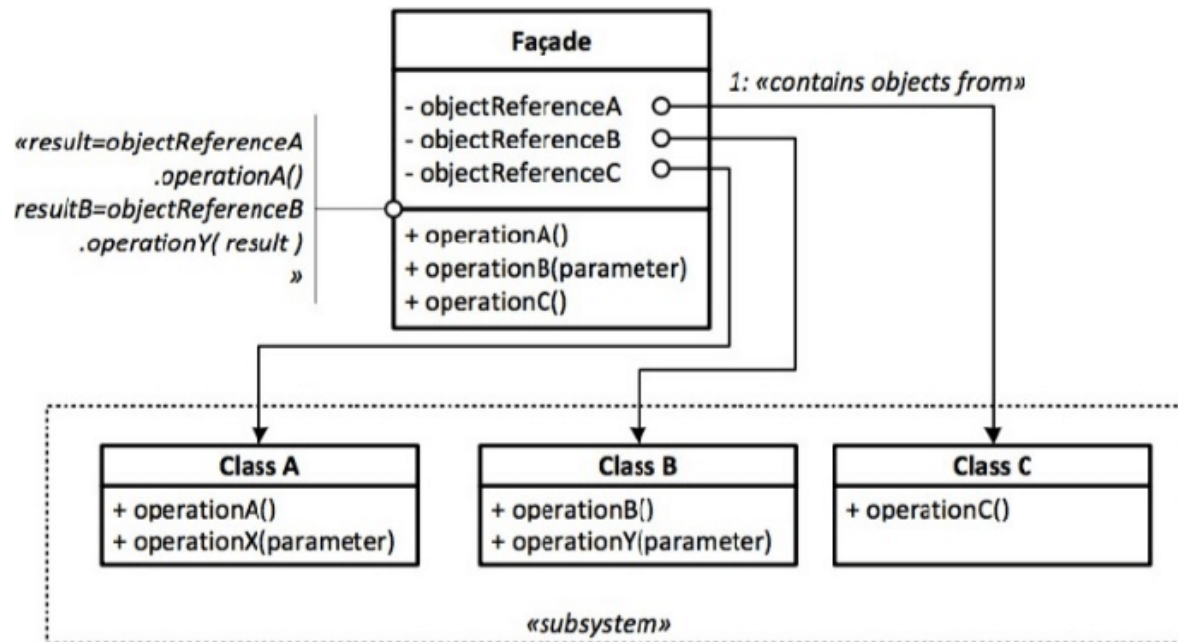


Pattern Façade

Façade est un patron de conception structurel qui procure une interface offrant un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.

- La Façade définit une nouvelle interface pour les objets existants, alors que l'Adaptateur essaye de rendre l'interface existante utilisable. L'*adaptateur* emballe généralement un seul objet alors que la *façade* s'utilise pour un sous-système complet d'objets.

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.
- Open/Closed Principle (OCP)
 - La classe est ouverte aux extensions cependant, elle peut ne pas être complètement fermée aux modifications si les composants internes du sous-système changent.
- 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 :

- Liskov Substitution Principle (LSP)
 - La Façade est généralement une classe unique qui encapsule plusieurs composants sans hériter d'autres classes, ce principe n'est pas directement applicable au pattern Façade.
- 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** : Bien que la Façade masque le sous-système, elle crée un point de dépendance unique. Si le sous-système change, la Façade doit être adaptée, ce qui entraîne un couplage indirect.
- **Complexité masquée** : En masquant la complexité, la Façade peut rendre plus difficile le débogage ou l'accès aux fonctionnalités avancées du sous-système, ce qui limite la flexibilité si des comportements spécifiques sont nécessaires.
- **Risque de surcharge** : Si la façade tente de couvrir trop de fonctionnalités, elle peut devenir trop complexe elle-même, annulant ainsi son but premier d'être une interface simplifiée.



Lien entre Décorateur et Façade

Le pattern **Adapter** est souvent lié au pattern **Décorateur**. La différence principale est que le **Décorateur** est utilisé pour **ajouter des fonctionnalités dynamiquement** à un objet sans changer son interface, tandis que l'**Adapter** modifie l'interface de l'objet pour qu'elle soit compatible avec une autre interface.

- Décorateur: Ne modifie pas l'interface mais ajoute une responsabilité (étend le comportement de l'objet)
- Adaptateur: Convertit une interface en une autre (rend l'objet utilisable dans un autre contexte)
- Façade: Simplifie une interface (masque la complexité)

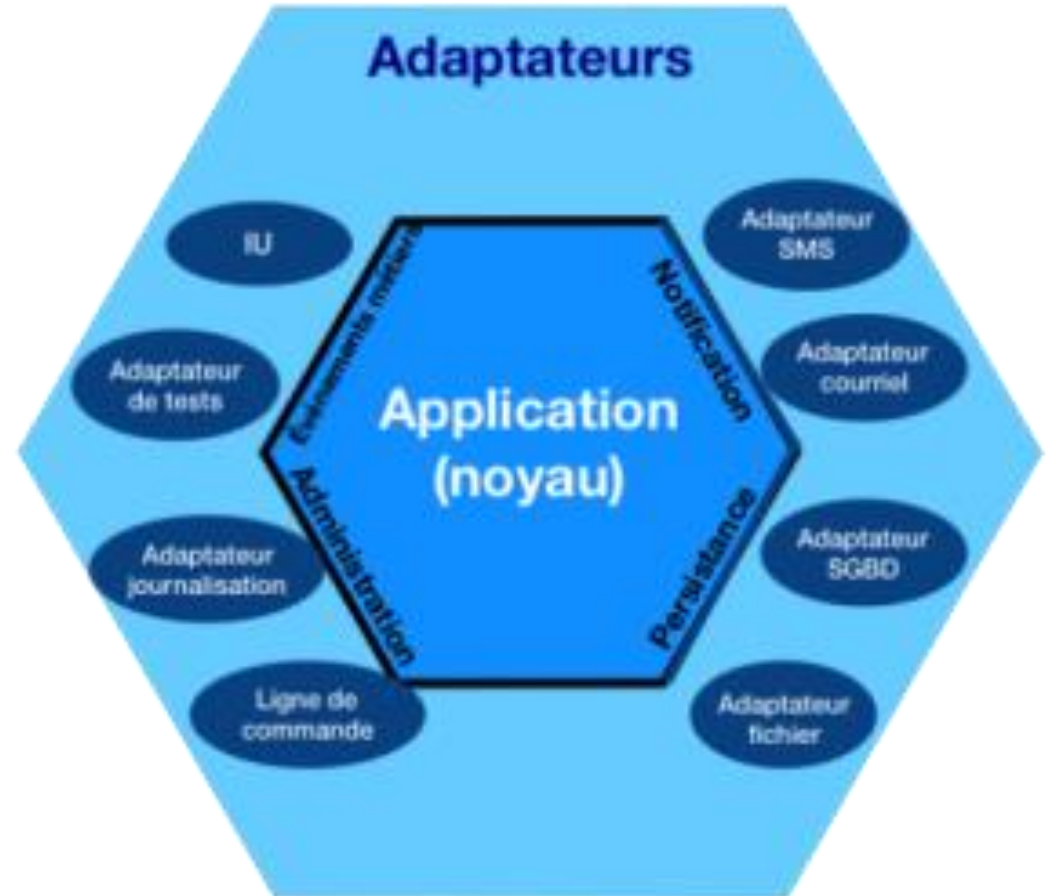
Les patterns **Adapter**, **Façade** et **Decorator** sont des solutions puissantes qui répondent à des besoins différents dans la conception logicielle. L'**Adapter** et le **Decorator** se concentrent sur la modification du comportement des objets, tandis que la **Façade** vise à simplifier l'interaction avec des systèmes complexes.

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/JyvRHxQMh8?si=F9rRlsutF3N6uaHS>



Live coding Pattern Adapter

Contexte :

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

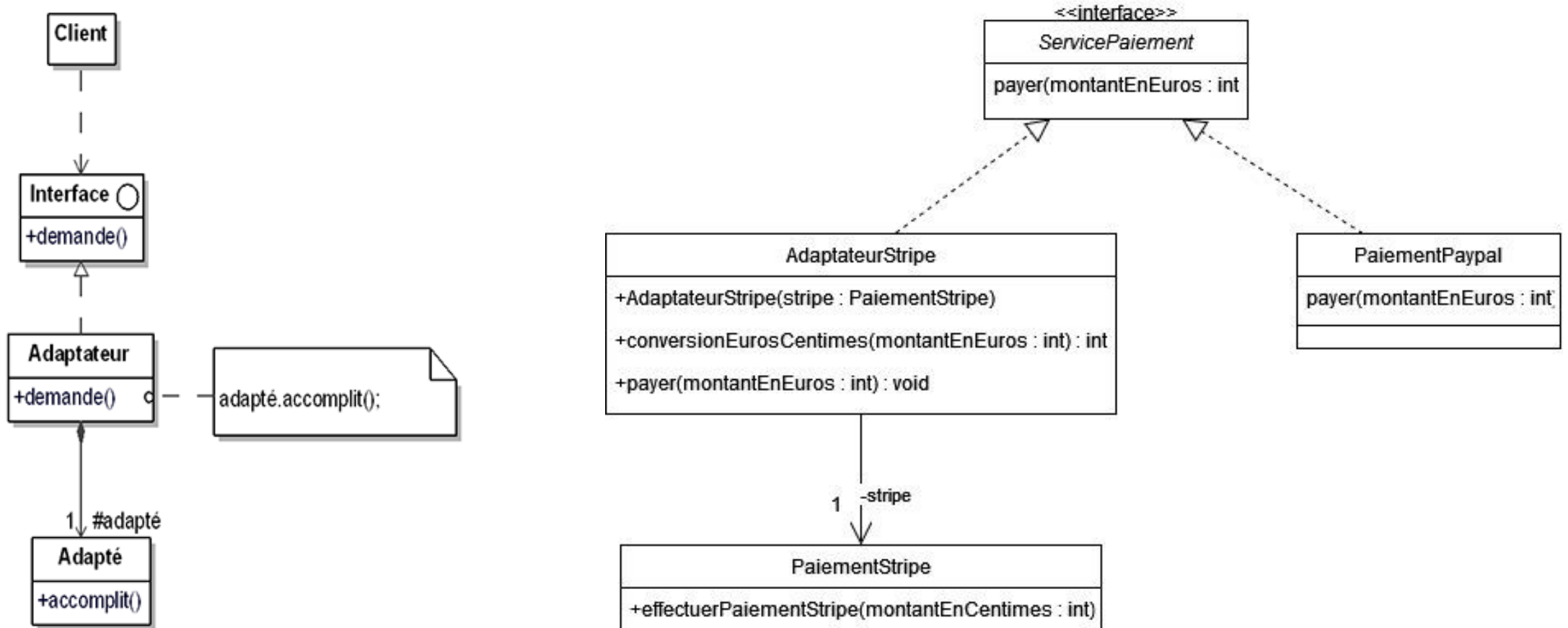
Problème :

Actuellement, les utilisateurs ne peuvent payer qu'avec PayPal, qui utilise des Euros. Cependant, nous aimerions diversifier les moyens de paiement en y intégrant Stripe. Le problème est que Stripe utilise des centimes comme format de données pour les paiements, ce qui est différent du format Euros de notre application.

Solution :

Pour résoudre ce problème, nous allons utiliser le pattern Adapter. Ce pattern nous permettra de convertir les données de Stripe (en centimes) au format Euros utilisé par notre application, facilitant ainsi l'intégration de Stripe tout en maintenant la cohérence du système de paiement.

Diagramme de classe Adapter





Live coding Pattern Façade



Contexte :

Notre application de commerce en ligne comprend plusieurs sous-systèmes pour gérer les commandes : la gestion de l'inventaire, le paiement et la logistique.

Problème :

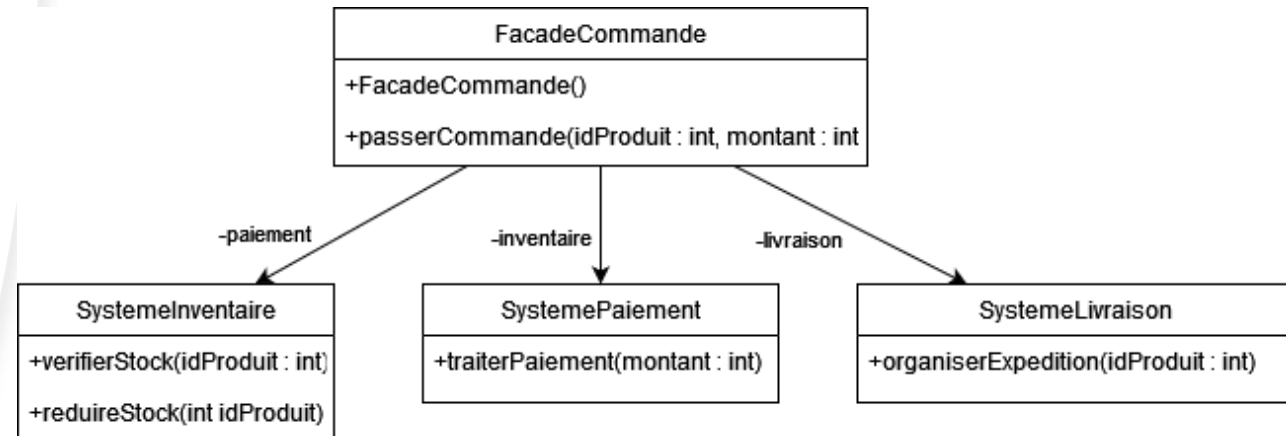
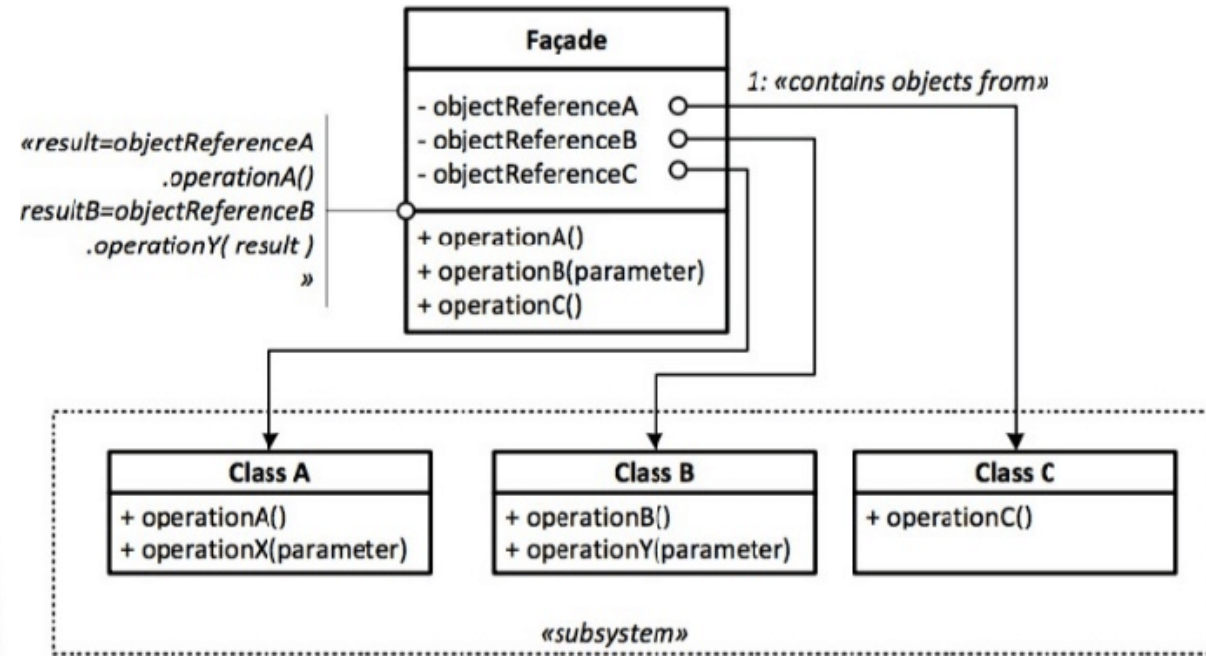
Actuellement, pour passer une commande, le client doit interagir avec chacun de ces sous-systèmes de manière indépendante, ce qui complique le code client et augmente le couplage entre les composants.

Solution :

Pour simplifier cette interaction et offrir un point d'entrée unique au client, nous allons utiliser le pattern Façade. La classe Façade encapsulera la complexité des appels aux différents sous-systèmes et présentera une interface unifiée.

Diagramme de classe Façade

27



Conclusion

En conclusion, les patterns **Adapter** et **Façade** apportent des solutions précieuses pour améliorer la flexibilité, la lisibilité et la maintenabilité d'un code en encapsulant des systèmes existants et en adaptant leurs interfaces aux besoins actuels.

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://www.sfeir.dev/back/les-design-patterns-structurels-facade/>



QCM