Pattern Adapter et Façade

HUGO BEQUET

ENZO DEGABRIEL

BEGUE MATTÉO

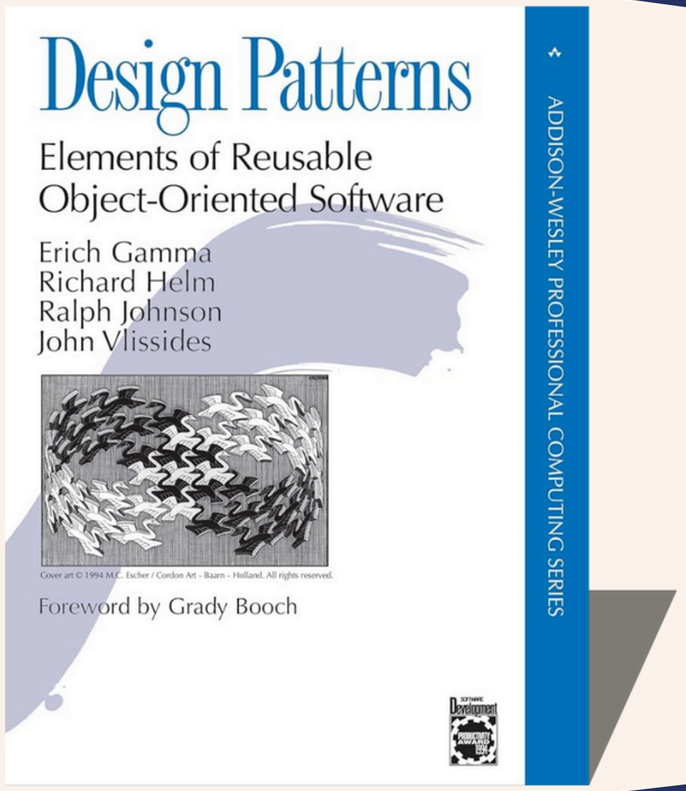
JULIE COURTY NGUYEN

SOMMAIRE

Présentation du GoF

Erich Gamma

Richard Helm



Ralph Johnson

John Vlissides

Présentation du pattern Adapter

Diagrammes génériques d'Adapter

Diagramme de classe

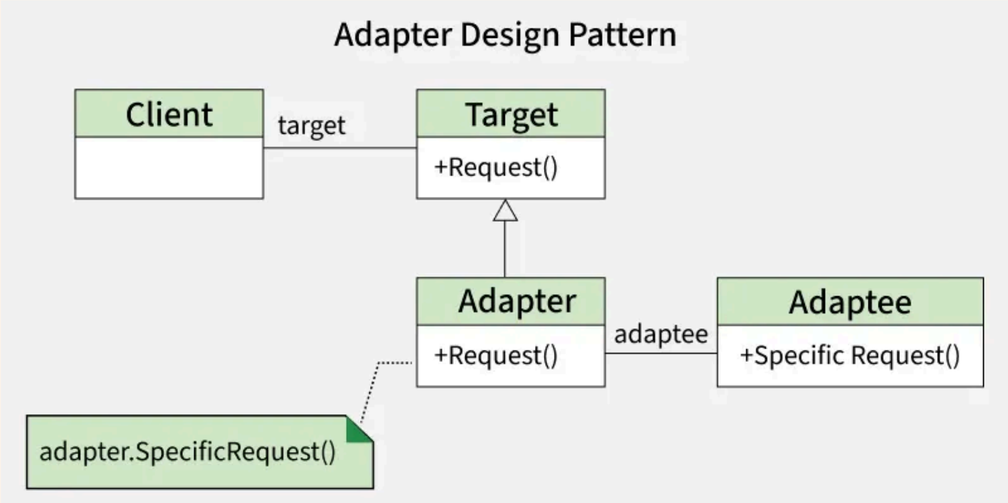
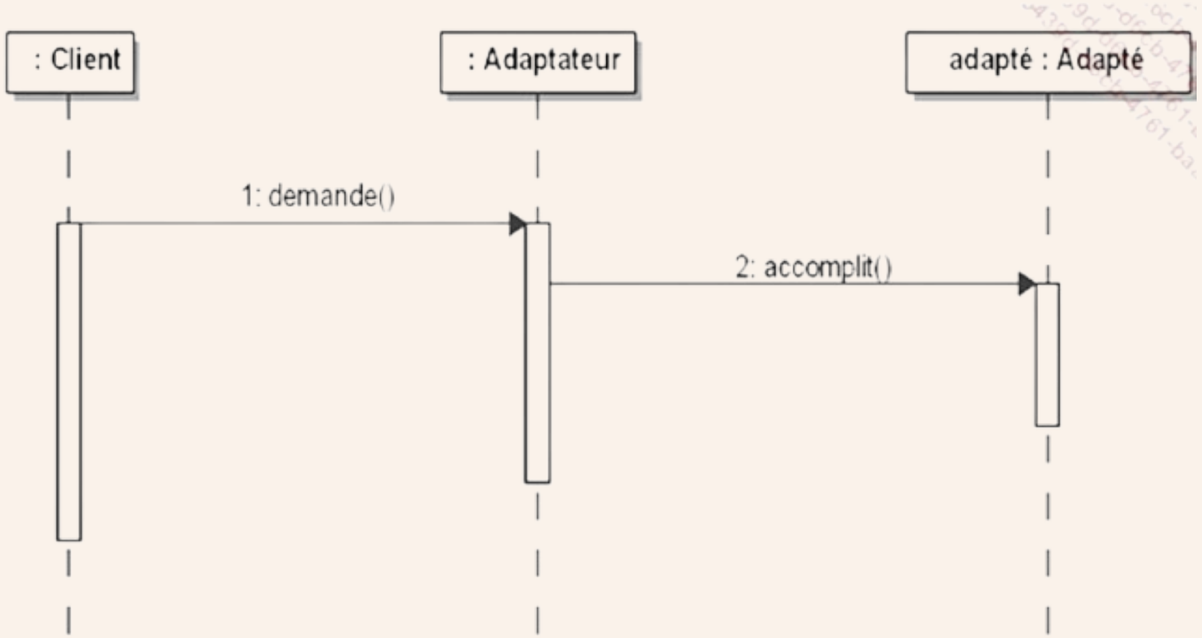


Diagramme de séquence



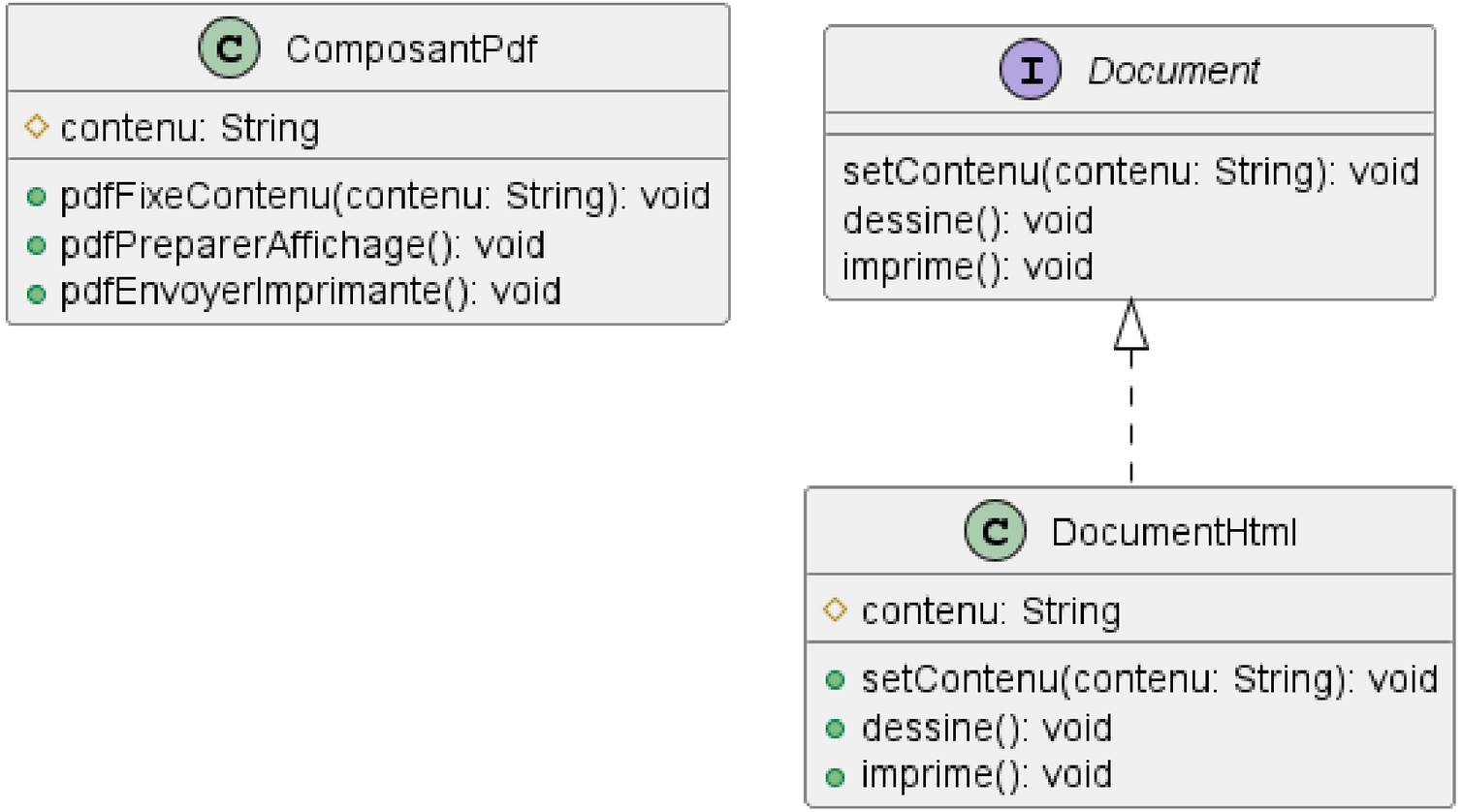
Exemple type de cas d'usage

```java
public interface Document {
    void setContenu(String contenu);
    void dessine();
    void imprime();
}
```

```java
public class ComposantPdf {
    protected String contenu;

    public void pdfFixeContenu(String contenu) {
        this.contenu = contenu;
    }

    public void pdfPreparerAffichage() {
        System.out.println("Affichage PDF : " + contenu);
    }

    public void pdfEnvoyerImprimante() {
        System.out.println("Impression PDF : " + contenu);
    }
}
```

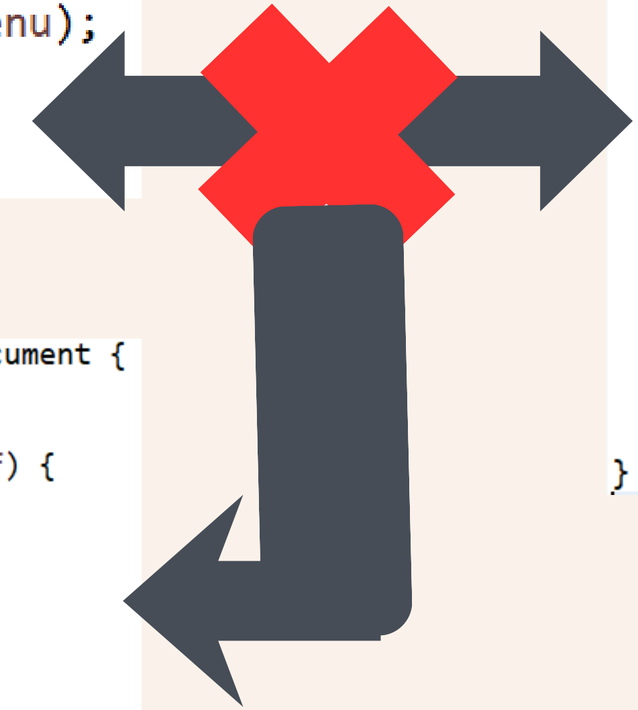
```java
public class DocumentHtml implements Document{
    protected String contenu;

    public void setContenu(String contenu) {
        this.contenu = contenu;
    }

    public void dessine() {
        System.out.println("Dessine document HTML : " + contenu);
    }

    public void imprime() {
        System.out.println("Imprime document HTML : " + contenu);
    }
}
```

Exemple type de cas d’usage



Exemple type de cas d’usage

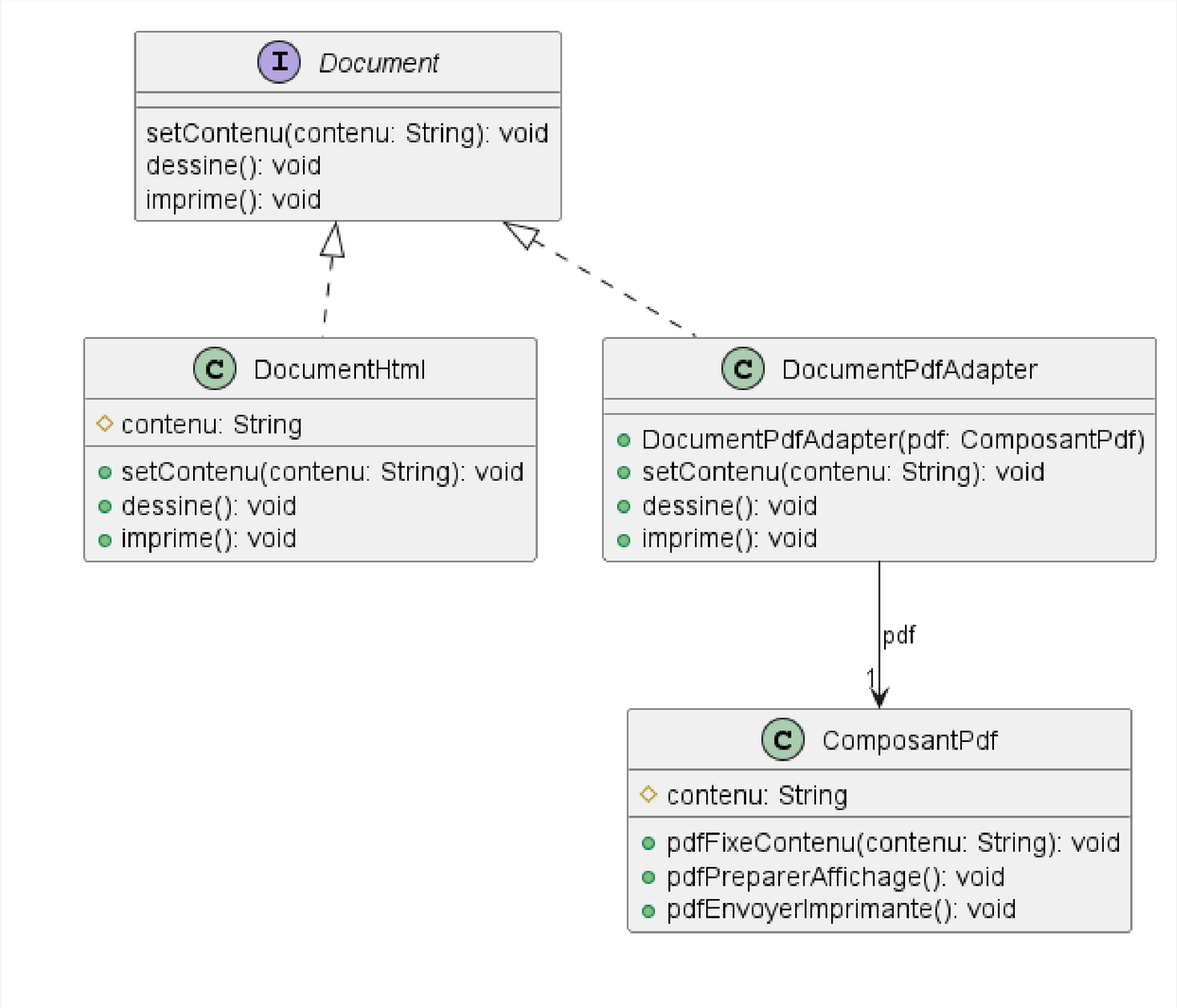
```java
public interface Document {
      void setContenu(String contenu);
      void dessine();
      void imprime();
}
```



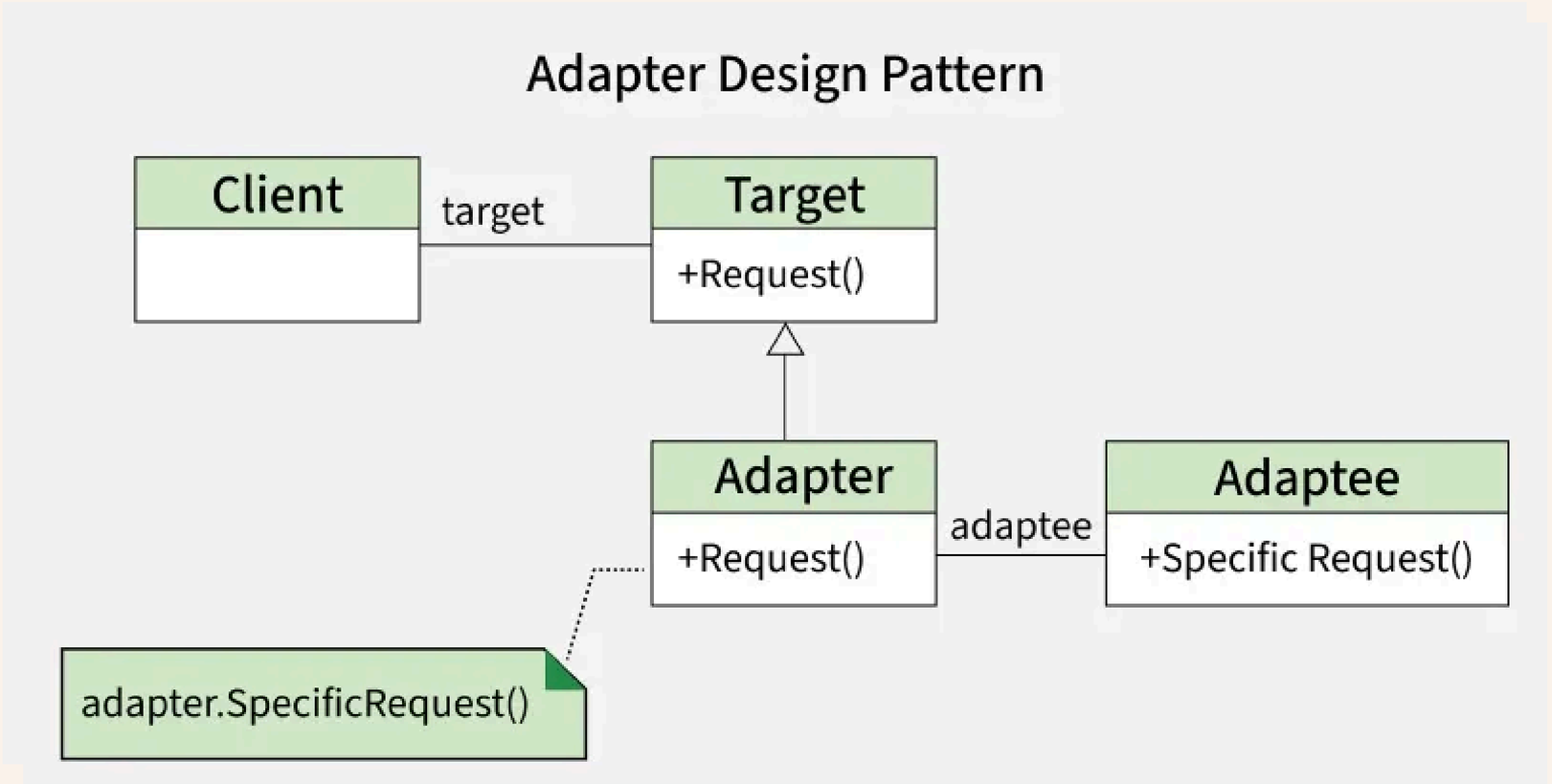
```java
public class DocumentPdfAdapter implements Document {
    protected ComposantPdf pdf;

    public DocumentPdfAdapter(ComposantPdf pdf) {
        this.pdf = pdf;
    }

    @Override
    public void setContenu(String contenu) {
        pdf.pdfFixeContenu(contenu);
    }

    @Override
    public void dessine() {
        pdf.pdfPreparerAffichage();
    }

    @Override
    public void imprime() {
        pdf.pdfEnvoyerImprimante();
    }
}
```

```java
public class ComposantPdf {
    protected String contenu;

    public void pdfFixeContenu(String contenu) {
        this.contenu = contenu;
    }

    public void pdfPreparerAffichage() {
        System.out.println("Affichage PDF : " + contenu);
    }

    public void pdfEnvoyerImprimante() {
        System.out.println("Impression PDF : " + contenu);
    }
}
```

Exemple type de cas d’usage



Classes participantes



Liens avec les principes SOLID

-SRP
L’Adapter a une seule responsabilité : convertir une interface en une autre.

-OCP
On peut ajouter de nouveaux Adapters sans modifier le code existant.

-LSP
L’Adapter peut remplacer n’importe quelle implémentation de l’interface cible sans problème.

-ISP
Le client utilise uniquement les méthodes dont il a besoin via une interface adaptée.

-DIP
Le client dépend d’une abstraction, pas d’une implémentation concrète.

Les limites du pattern

Complexité inutile si les interfaces sont très proches
Adapter des interfaces très proches peut compliquer le code sans réel bénéfice (ex. prise téléphone vs prise PC)

Multiplication des Adapters
Plusieurs systèmes incompatibles nécessitent plusieurs Adapters, ce qui alourdit la maintenance

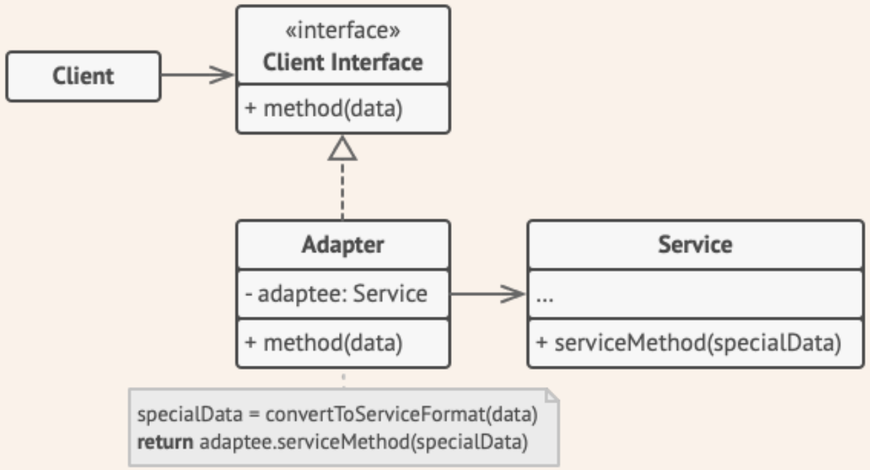
Peut cacher un mauvais design
Enchaîner plusieurs Adapters complique la compréhension et le débogage du code

Risque de créer une chaîne d'adaptations
L'Adapter corrige les symptômes mais peut cacher des problèmes d'architecture plus profonds

Lien entre Adapter et Decorator

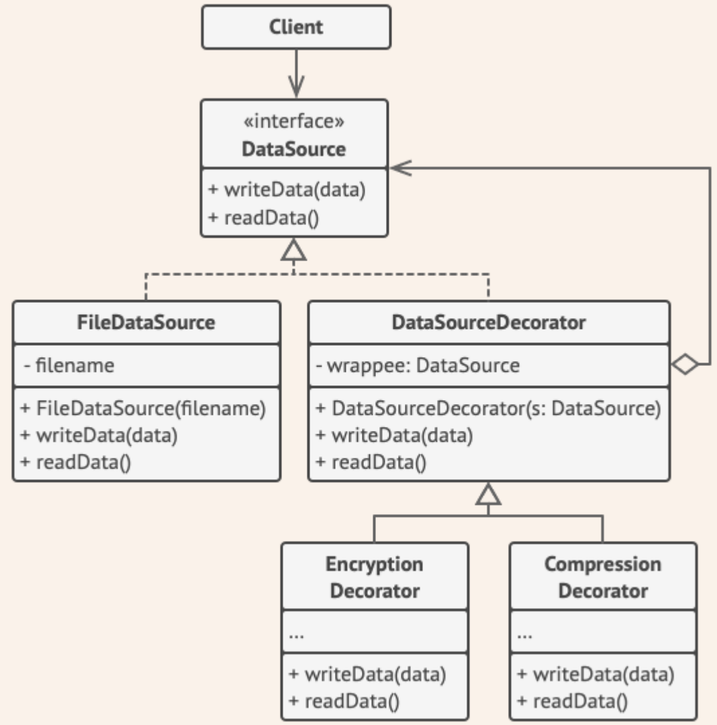
Adapter

Transforme une interface existante pour la rendre compatible avec ce que le client attend



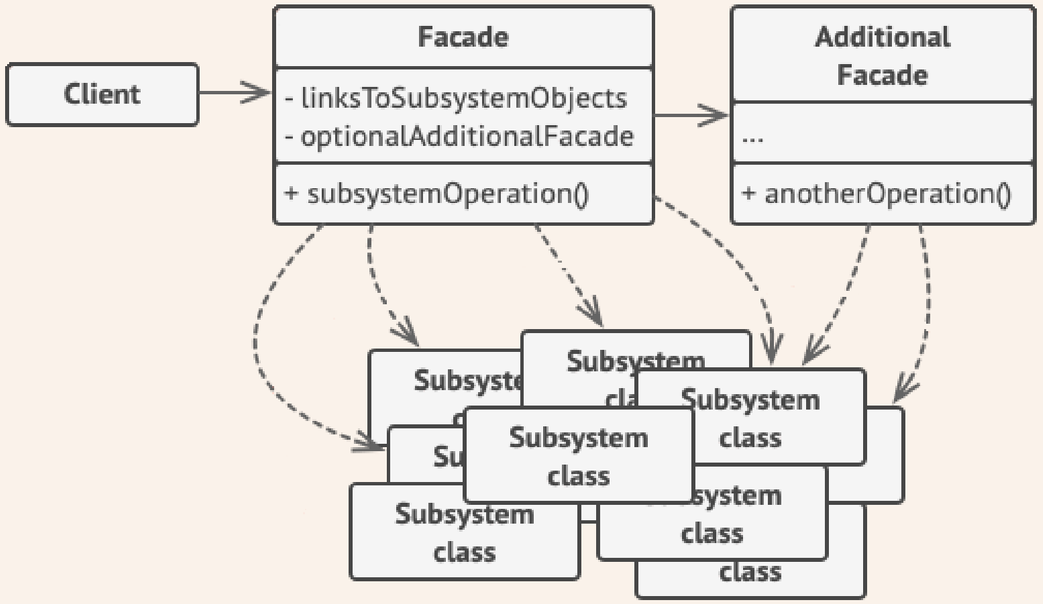
Decorator

Ajoute des fonctionnalités supplémentaires à un objet, sans changer son interface



Présentation du pattern Façade

Diagramme de classe



Exemple type de cas d'usage

```java
public class Projector {

    public void on() {
        System.out.println("Projector on");
    }

    public void setInput(String input) {
        System.out.println("Projector input set to: "+ input);
    }

    public void off() {
        System.out.println("Projector off");
    }
}
```

```java
public class SoundSystem {

    public void on() {
        System.out.println("Sound System on");
    }

    public void setVolume(int level) {
        System.out.println("Sound System volume set to " + level);
    }

    public void off() {
        System.out.println("Sound System off");
    }
}
```

```java
public class Screen {
    public void down() {
        System.out.println("Screen is down");
    }

    public void up() {
        System.out.println("Screen is up");
    }
}
```

```java
public class DvdPlayer {
    public void on() {
        System.out.println("Dvd Player on");
    }

    public void play(String movie) {
        System.out.println("Playing movie :" + movie );
    }

    public void off() {
        System.out.println("Dvd Player off");
    }
}
```

Exemple type de cas d’usage

```java
public class HomeTheaterFacade {
    private DvdPlayer dvdPlayer;
    private SoundSystem soundSystem;
    private Projector projector;
    private Screen screen;

    public HomeTheaterFacade() {
        this.dvdPlayer = new DvdPlayer();
        this.soundSystem = new SoundSystem();
        this.projector = new Projector();
        this.screen = new Screen();
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        screen.down();
        projector.on();
        projector.setInput("DVD");
        soundSystem.on();
        soundSystem.setVolume(10);
        dvdPlayer.on();
        dvdPlayer.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting movie theater down...");
        dvdPlayer.off();
        soundSystem.off();
        projector.off();
        screen.up();
    }
}
```

```java
public class FacadePatternDemo {
    public static void main(String[] args) {
        HomeTheaterFacade homeTheaterFacade = new HomeTheaterFacade();

        homeTheaterFacade.watchMovie("Inception");
        homeTheaterFacade.endMovie();
    }
}
```

Liens avec les principes SOLID

-SRP
la Façade a une seule responsabilité : simplifier l’accès à un sous-système complexe.

-OCP
Le pattern peut être développé sans modifier les classe internes.

-LSP
Pas d’héritage présent.

-ISP
Une interface qui peut rapidement devenir trop large.

-DIP
Dépend généralement de classes concrètes et non abstraites.

Les limites du pattern

Surchage de la nôtre Facade
La classe peut rapidement devenir trop complexe
en récupérant tout.

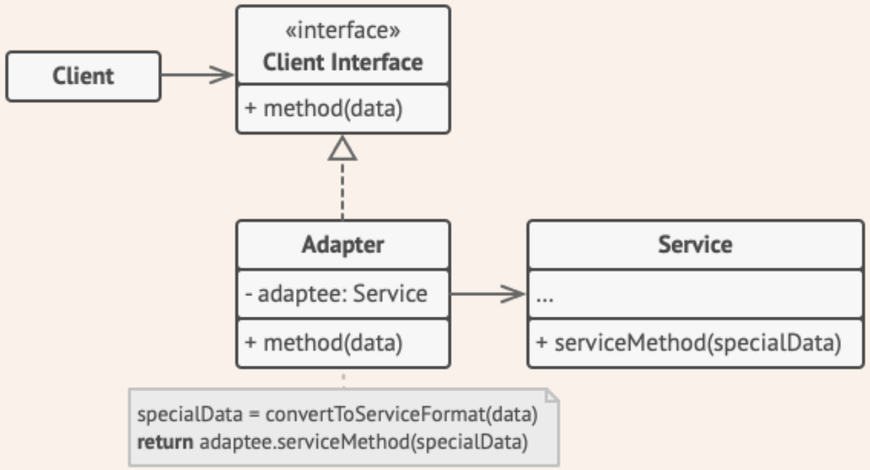
Complexité architecturale
Créer plusieurs Facade pour éviter une surcharge
peut entrainer une création excessive de classe.

Risque de Duplication
Plus on crée de Facade, plus les créations de
comportements similaires deviennent facile.

Lien entre Adapter et Facade

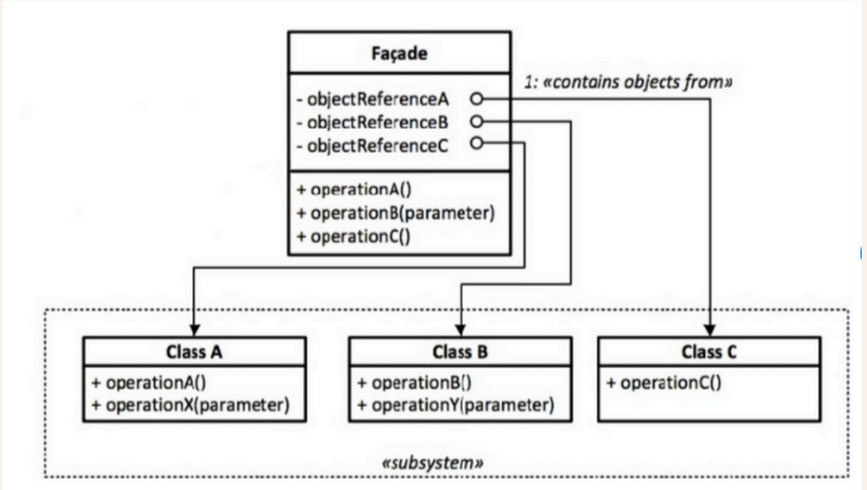
Adapter

Transforme une interface existante pour la rendre compatible avec ce que le client attend



Facade

Définit une nouvelle interface d'objets existants pour en simplifier l'accès



Live coding

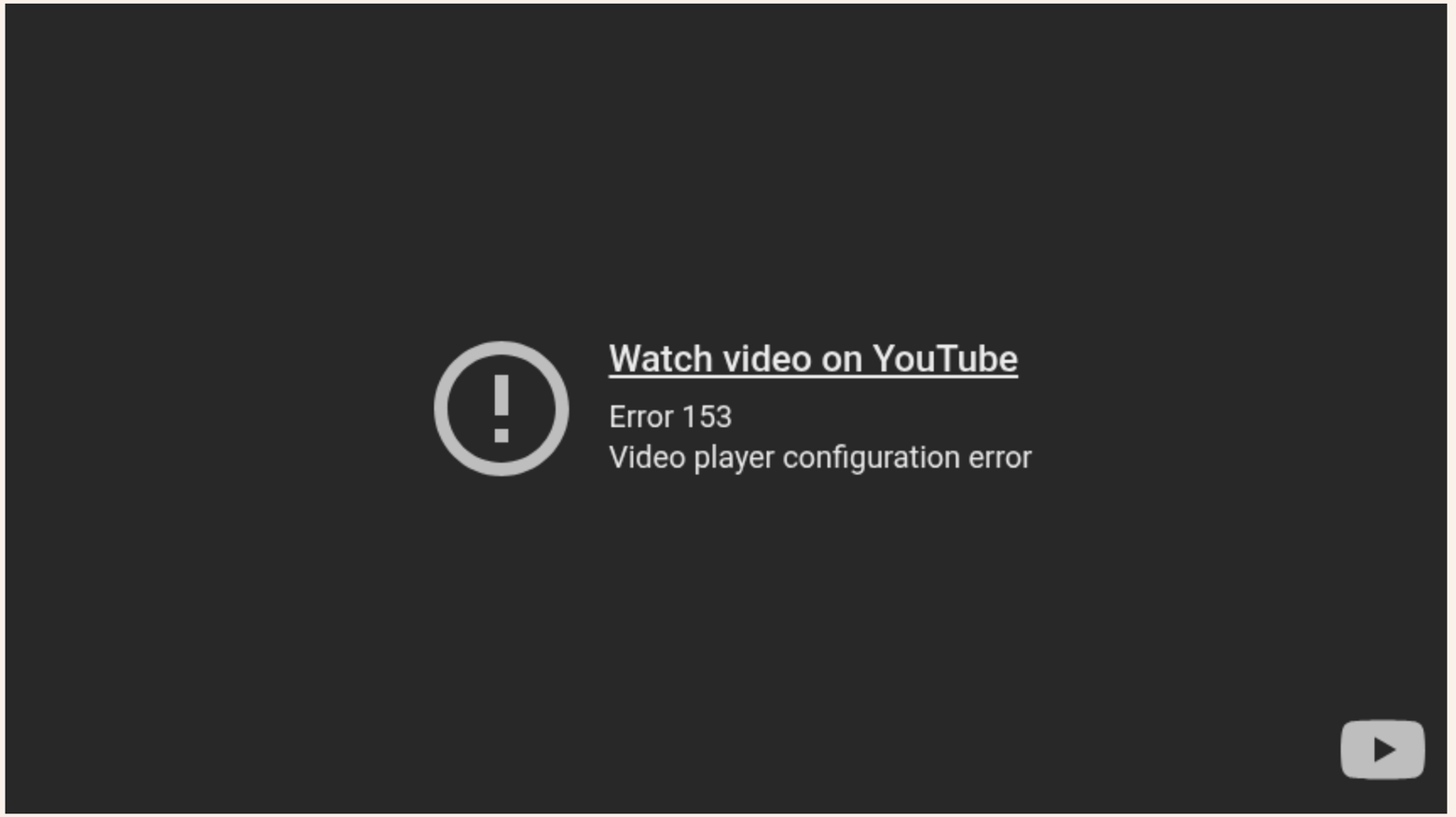
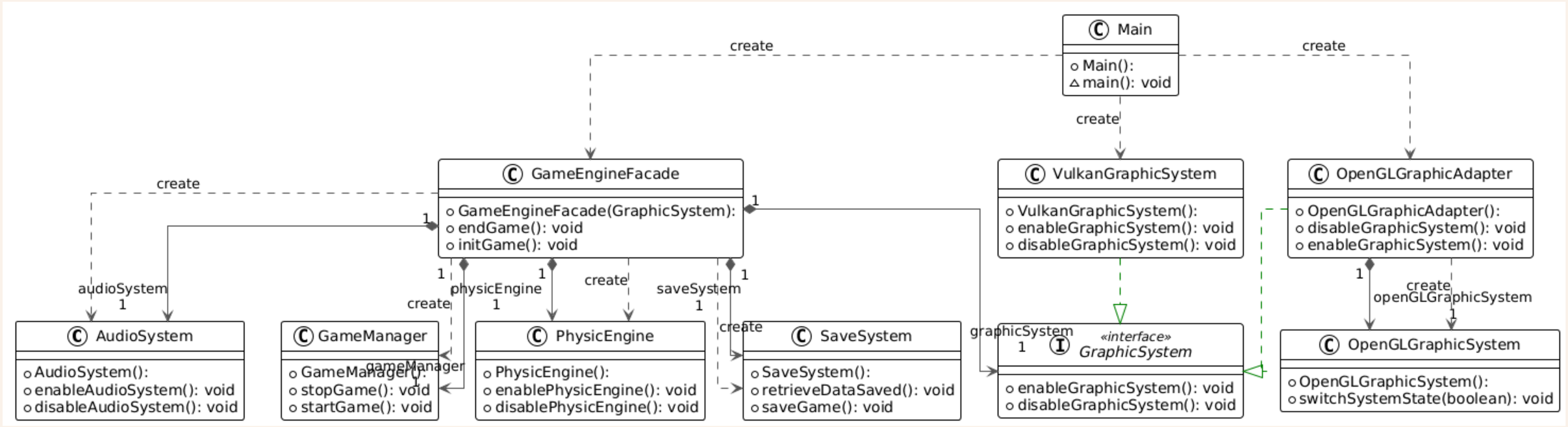
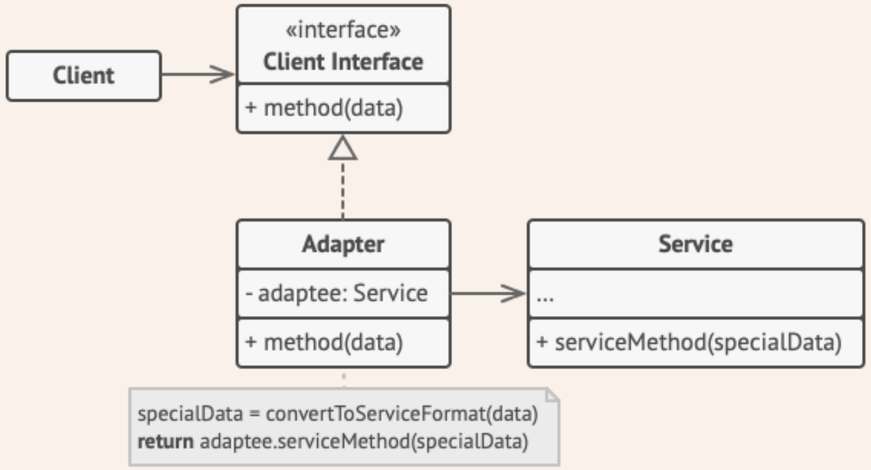
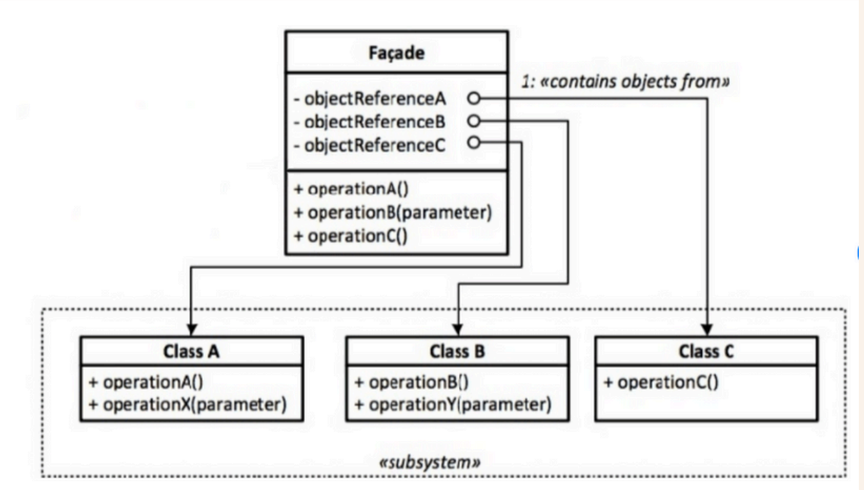

Watch video on YouTube
Error 153
Video player configuration error

Diagramme de classe final







Conclusion

Bibliographie

https://refactoring.guru/design-patterns/adapter

https://refactoring.guru/design-patterns/facade

https://www.bob-le-developpeur.com/notions/design-patterns

Youtube : The Adapter Pattern Explained and Implemented in Java | Structural Design Patterns | Geekific

https://cdiese.fr/design-pattern-facade/

https://copilot.microsoft.com/chats/FAvihHgGSTkScGGH16ZsT

QCM