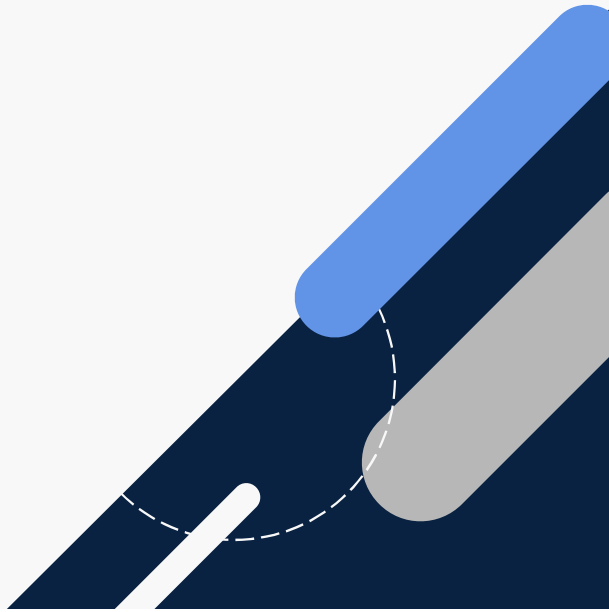
Pattern Bridge



Définition

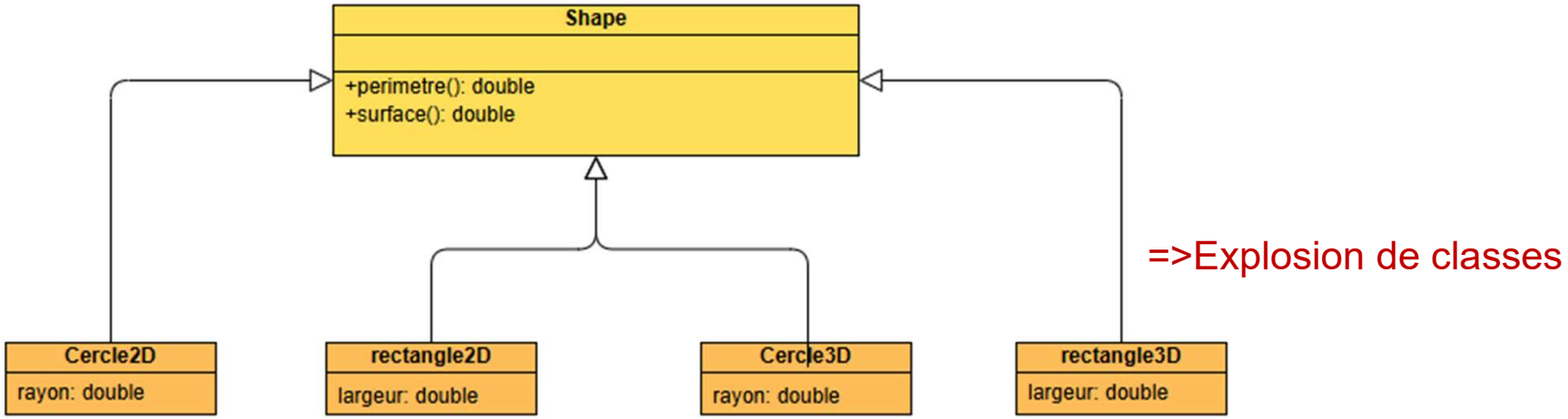
- Le pattern Bridge a pour but de dissocier une abstraction de son implémentation, afin qu'elles puissent évoluer indépendamment l'une de l'autre.

- En d'autres termes, on veut éviter de combiner plusieurs dimensions de variation pour une même hiérarchie de classes.

Problème typique

Supposons qu'on ait une classe Shape et des sous-classes comme Circle, Square. Et que chaque forme puisse être dessinée soit en 2D, soit en 3D.

Sans Bridge, on aurait cette structure UML:



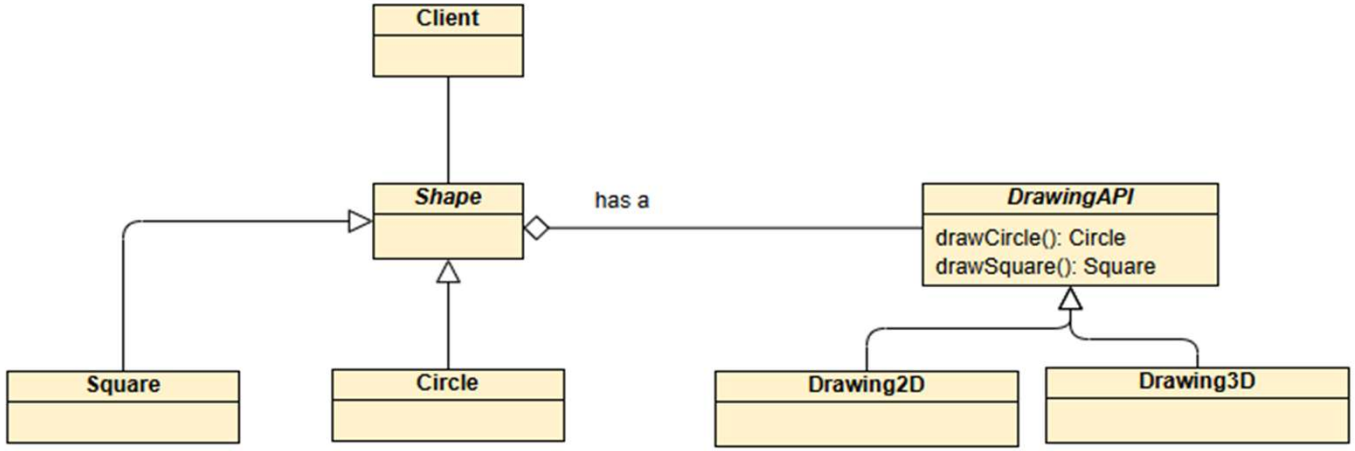
=>Explosion de classes

Solution Bridge

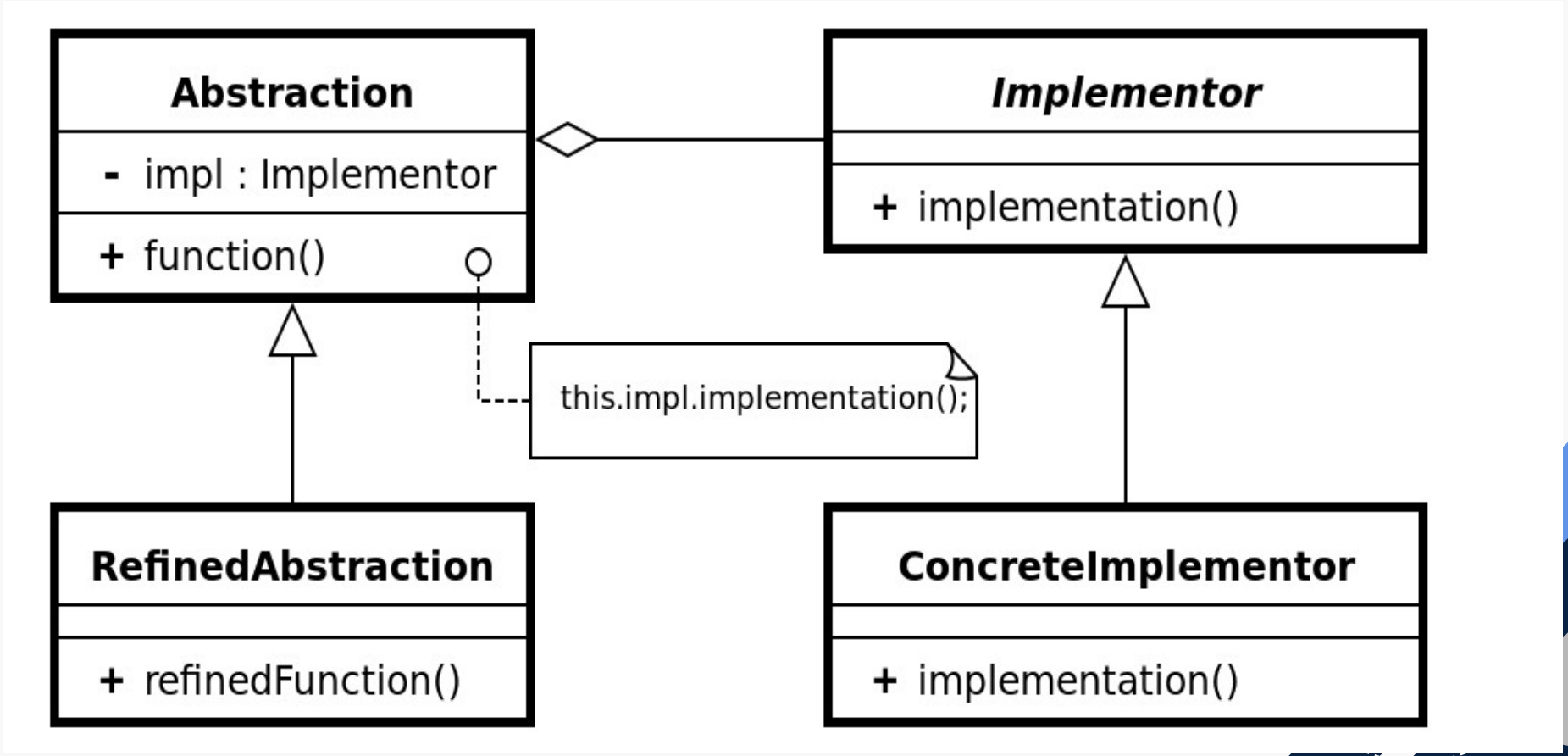
Avec Bridge, on **sépare les deux dimensions d'évolution** :

Dimension 1	Dimension 2
Type de forme (abstraction)	Type de rendu (implémentation)

On obtient la structure suivante:



Structure UML



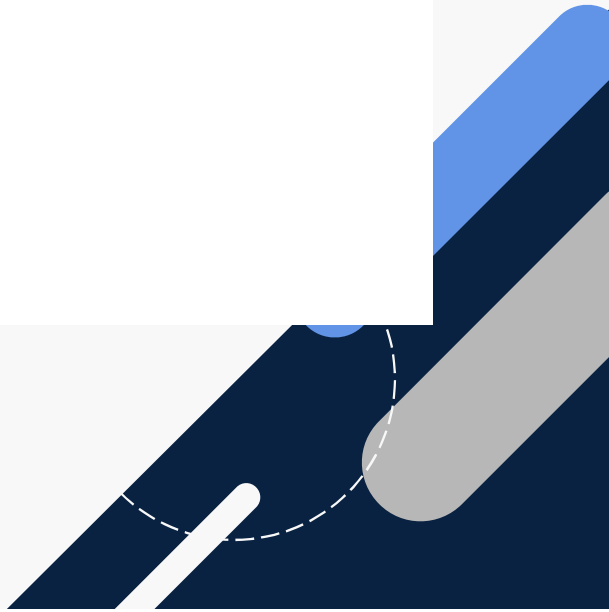
Avantages et inconvénients

Avantages:

- Évite la prolifération de classes.
- Abstraction et implémentation évoluent indépendamment.
- Favorise la composition plutôt que l'héritage.

Inconvénients:

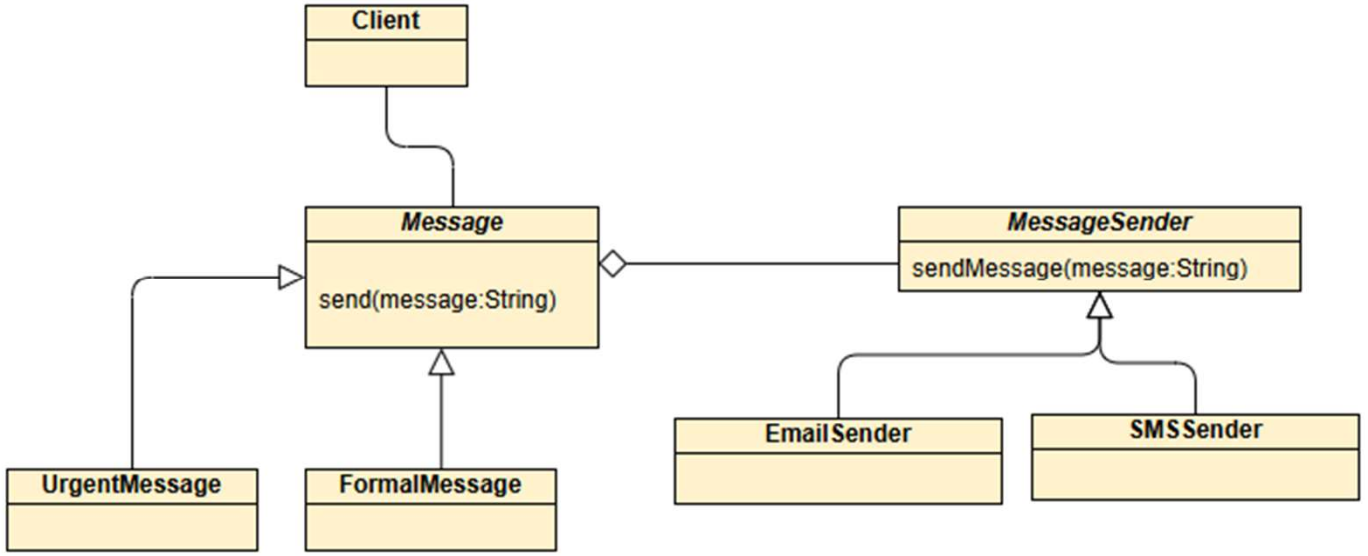
- Plus de classes et d'indirections.
- Complexité accrue pour les petits projets.



Exercice 1— Messages avec différents canaux

Objectif:

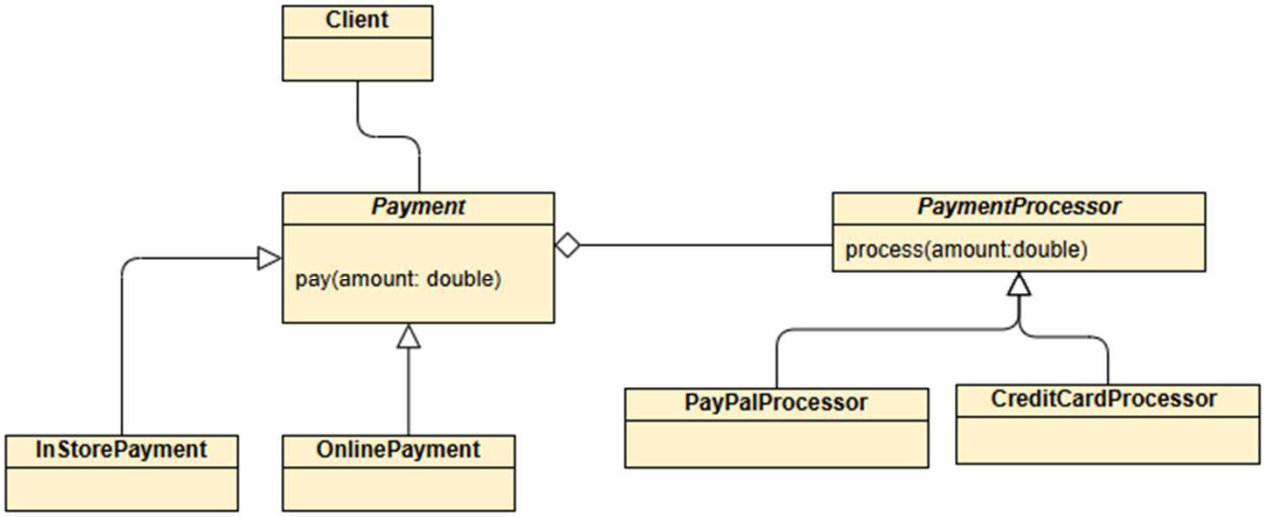
Créer un système où un Message peut être envoyé via différents canaux (Email, SMS, Push).



Exercice 2— Paiement en ligne

Objectif:

Créer une hiérarchie Payment indépendante du mode de paiement (PayPal, CreditCard).



Exercice de synthèse — Système de génération de rapports multi-format

Objectif:

Une société d'analyse de données veut créer un système de génération automatique de rapports. Chaque type de rapport (ex : Rapport financier, Rapport de performance, Rapport RH) peut être exporté dans plusieurs formats : HTML, PDF, Excel.

Actuellement, le système est conçu de manière naïve :FinancialReportPDF, FinancialReportHTML, FinancialReportExcelPerformanceReportPDF, PerformanceReportHTML, PerformanceReportExcelHRReportPDF, HRReportHTML, HRReportExcel

Exercice de synthèse — Système de génération de rapports multi-format

Travail demandé:

1. Identifiez les problèmes de la solution actuelle
2. Identifiez les deux dimensions d'évolution présentes dans ce système.
3. Proposez une refactorisation qui permette :

- d'ajouter un nouveau type de rapport (ex : Rapport d'audit) sans modifier les formats ;
- d'ajouter un nouveau format (ex : JSON) sans modifier les rapports existants.

4. Donnez un diagramme de classes conceptuel de la nouvelle architecture.

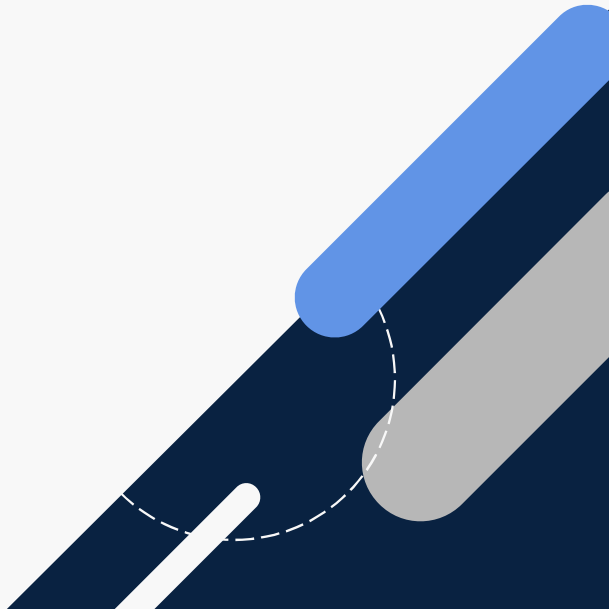
5. Proposez une illustration en pseudo-code ou Java simpli

Exercice de synthèse — Système de génération de rapports multi-format

Problèmes de la solution actuelle :

- Chaque nouvelle combinaison type de rapport × format de sortie entraîne la création d'une nouvelle classe.
- Les changements dans un format (ex : PDF → nouvelle librairie) imposent des modifications dans toutes les classes concernées.
- Il devient impossible de faire évoluer le système sans le casser.

Pattern
Proxy



Définition

- Le design pattern Proxy est un modèle de conception structurel qui fournit un substitut ou un intermédiaire à un autre objet pour contrôler l'accès à cet objet.
- En Java, un proxy peut être utilisé pour ajouter une couche d'indirection et d'interception, permettant de modifier ou de contrôler l'accès aux fonctionnalités d'un objet sans en altérer le code source.

Problème typique

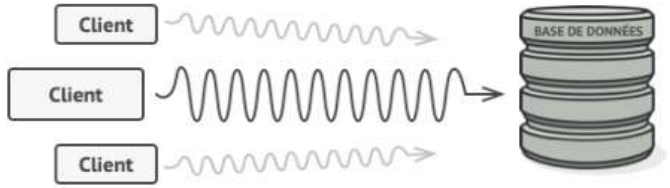
On dispose d'un objet RealSubject coûteux à créer ou à manipuler.

On désire:

- Retarder la création jusqu'au moment où il est vraiment nécessaire (lazy loading).
- Contrôler l'accès selon l'utilisateur ou le contexte (protection).
- Ajouter des fonctionnalités comme le logging ou la mise en cache.

Problème typique

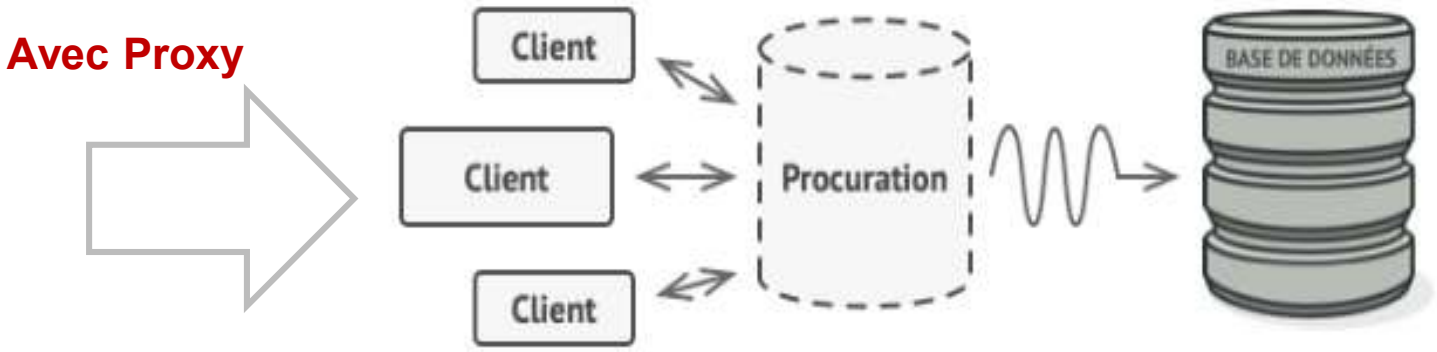
Exemple:



Les requêtes sur la base de données sont parfois très lentes.

Sans Proxy, chaque client manipule directement l'objet réel et toutes les fonctionnalités supplémentaires doivent être intégrées dans l'objet.

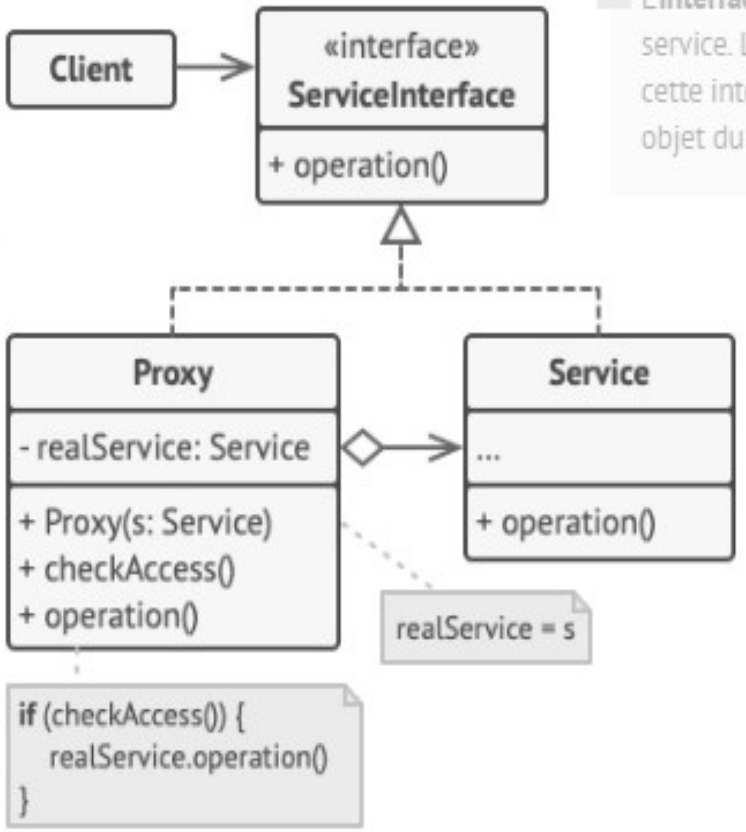
Problème typique



- Proxy propose de créer une classe procuration qui a la même interface que l'objet du service original.
- L'objet procuration est passé à tous les clients de l'objet original.
- Lors de la réception d'une demande d'un client, la procuration crée l'objet du service original et lui délègue la tâche.

Structure UML

4 Le **Client** passe par la même interface pour travailler avec les services et les procurations. Il est ainsi possible de passer une procuration à n'importe quel code qui attend un objet service.



1 L'**Interface Service** déclare l'interface du service. La procuration doit implémenter cette interface afin de pouvoir se déguiser en objet du service.

3 La **Procuration** est une classe dotée d'un attribut qui pointe vers un objet service. Une fois que la procuration a lancé tous ses traitements (instanciation paresseuse, historisation des logs, vérification des droits, mise en cache, etc.), elle envoie la demande à l'objet du service.

En général, les procurations gèrent le cycle de vie de leurs objets service.

2 Le **Service** est une classe qui fournit la logique métier dont vous voulez vous servir.

Types de proxy

Type	Description
Virtual Proxy	Retarde la création d'un objet coûteux jusqu'à ce qu'il soit réellement utilisé.
Protection Proxy	Contrôle l'accès à l'objet selon les droits de l'utilisateur.
Remote Proxy	Permet d'accéder à un objet situé sur un autre serveur/machine.
Cache Proxy	Stocke les résultats d'appels coûteux pour éviter de recalculer.
Logging Proxy	Intercepte les appels pour journaliser ou monitorer l'utilisation.

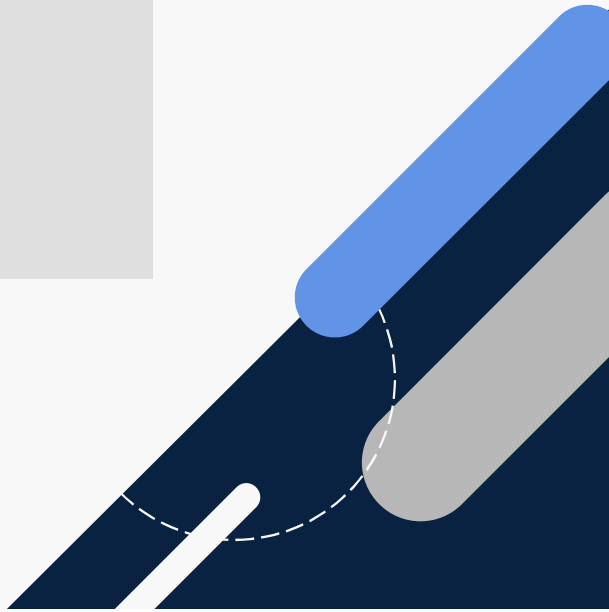
Exemple 1— Virtual Proxy

```
// Interface commune
interface Image {
    void display();
}
// Objet réel
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }
    private void loadFromDisk() {
        System.out.println("Chargement de " + filename);
    }
    public void display() {
        System.out.println("Affichage de " + filename);
    }
}
```

```
// Proxy
class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
// lazy loading
        }
        realImage.display();
    }
}
```

Exemple 1— Virtual Proxy

```
// Test
public class ProxyDemo {
    public static void main(String[] args) {
        Image img1 = new ProxyImage("photo1.jpg");
        Image img2 = new ProxyImage("photo2.jpg");

        img1.display(); // charge et affiche
        img1.display(); // affiche sans recharger
        img2.display(); // charge et affiche
    }
}
```



Avantages et inconvénients

Avantages:

- Contrôle fin sur l'accès à l'objet réel.
- Permet le lazy loading, le caching, la sécurité, le logging, etc.
- Le client ne voit pas la différence entre le Proxy et l'objet réel.

Inconvénients:

- Complexifie légèrement le code.
- Peut introduire une surcharge si le Proxy ajoute beaucoup de logique.

Exercice 1

Une entreprise veut protéger l'accès à certaines fonctionnalités selon le rôle utilisateur (Admin, Guest).

Les composants utilisés :

- Account : interface avec la méthode access().
- RealAccount : implémentation réelle des fonctionnalités.
- AccountProxy : contrôle si l'utilisateur a les droits pour accéder.

Questions:

1. Quel est le rôle du Proxy dans ce scénario ?
2. Quelle est la différence entre RealAccount et AccountProxy?

Exercice 1

Conception:

1. Dessinez un diagramme de classes montrant les relations entre Account, RealAccount et AccountProxy.
2. Expliquez comment vous pouvez ajouter un nouveau rôle utilisateur (Manager) sans modifier RealAccount.

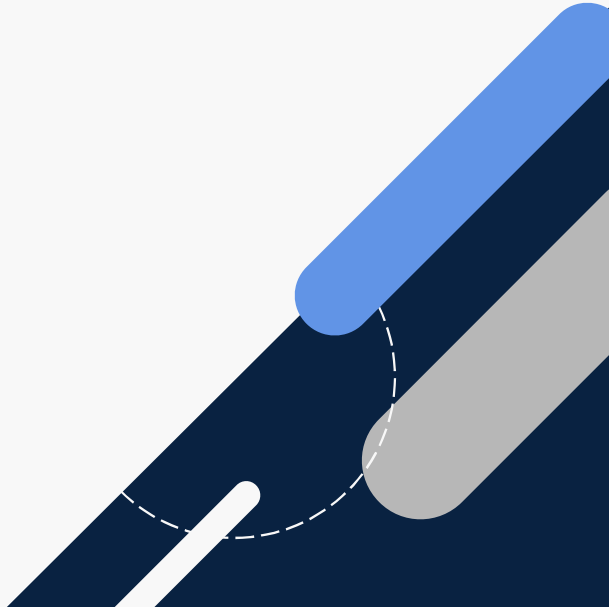
Implémentation

1. Implémentez l'interface Account.
2. Implémentez RealAccount pour afficher "Accès accordé" lors d'un accès.
3. Implémentez AccountProxy pour vérifier le rôle de l'utilisateur et refuser l'accès si le rôle n'est pas autorisé.

Exercice 1

Test

Créez des comptes pour Admin et Guest et testez access() pour chaque rôle.



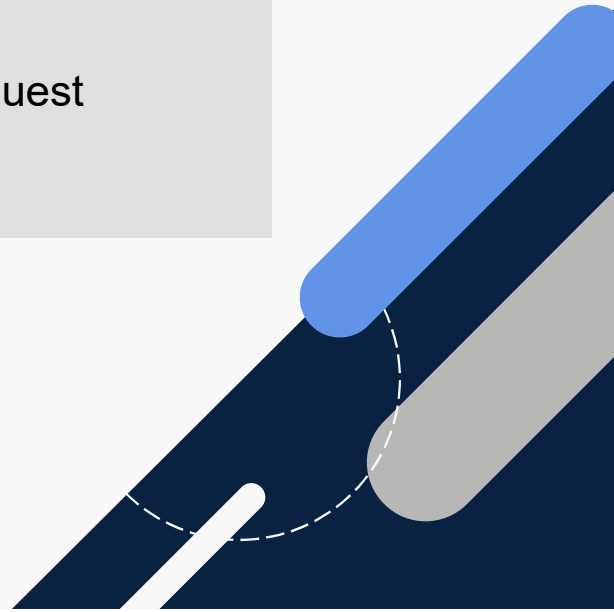
Exercice 1

```
// 1. Interface commune
interface Account {
    void access();
}

// 2. Objet réel
class RealAccount implements Account {
    public void access() {
        System.out.println("Accès accordé");
    }
}
```

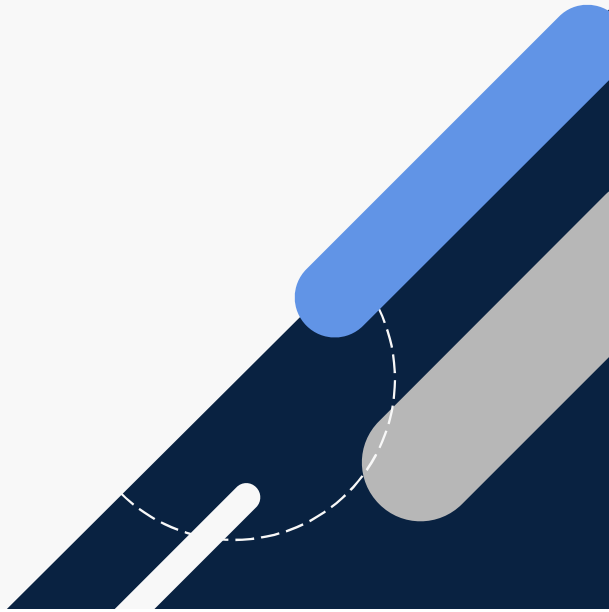
```
// 3. Proxy
class AccountProxy implements Account {
    private RealAccount realAccount;
    private String role;

    public AccountProxy(String role) {
        this.role = role;
    }

    public void access() {
        if (role.equalsIgnoreCase("Admin")) {
            if (realAccount == null) {
                realAccount = new RealAccount(); // lazy
instantiation
            }
            realAccount.access();
        } else {
            System.out.println("Accès refusé pour le rôle : " +
role);
        }
    }
}
```

Exercice 1

```
// 4. Test
public class ProxyAccessTest {
    public static void main(String[] args) {
        Account adminAccount = new AccountProxy("Admin");
        Account guestAccount = new AccountProxy("Guest");

        adminAccount.access();  // Affiche: Accès accordé
        guestAccount.access();  // Affiche: Accès refusé pour le rôle : Guest
    }
}
```



Pattern Flyweight



Définition

- Le pattern Flyweight est un patron de structure qui permet de partager des objets similaires afin de réduire la consommation mémoire et d'améliorer les performances.
- Il sépare les informations d'un objet en :
  - État intrinsèque : commun, partagé entre plusieurs objets (stocké dans le Flyweight).
  - État extrinsèque : propre à chaque contexte, non partagé (fourni lors de l'utilisation).

Différence entre État Intrinsèque et État Extrinsèque

Type d'état	Définition	Stockage	Partage	Exemple général
État intrinsèque	Ensemble des informations **internes et invariables** de l'objet, **communes** à plusieurs instances.	Dans le **Flyweight** lui-même.	partagé entre plusieurs clients.	Forme, texture, police, type, configuration fixe.
État extrinsèque	Informations **contextuelles et changeantes** dépendant de **l'utilisation** de l'objet.	Dans le **client** (ou passé en paramètre).	propre à chaque usage.	Position, couleur, état, coordonnées, vitesse.

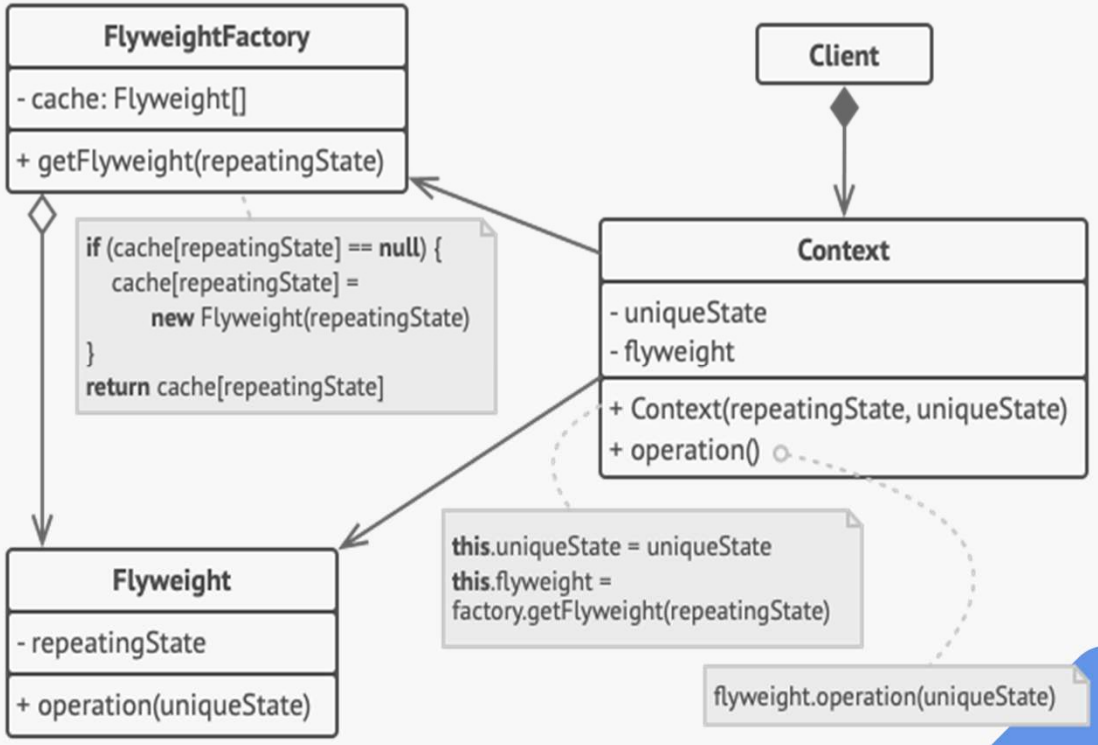
Exemple de voiture de location

- Imaginons une société de location avec 100 voitures du même modèle :

Élément	Type d'état	Pourquoi
Modèle, marque, puissance moteur, couleur du châssis	**Intrinsèque**	Ce sont des caractéristiques communes, identiques pour toutes les voitures de ce modèle.
Plaque d'immatriculation, kilométrage, conducteur actuel, destination	**Extrinsèque**	Ce sont des informations propres à chaque voiture au moment de l'utilisation.

- Le Flyweight correspond à "la fiche technique du modèle de voiture".
- L'état extrinsèque est géré par le "client" (la location courante).

Structure UML



Ce diagramme montre les trois acteurs principaux du pattern Flyweight :

- ❑ **FlyweightFactory** → fabrique et gère les objets partagés (cache).
- ❑ **Flyweight** → représente l'objet léger contenant l'état intrinsèque.
- ❑ **Context** → représente chaque utilisation spécifique, contenant l'état extrinsèque.
- ❑ **Client** → demande la création/utilisation d'un objet via la Factory.

Le but est de **réduire la duplication** en **partageant** l'état commun entre plusieurs contextes.

Exemple : Editeur de texte

- Dans un éditeur, chaque caractère (A, B, C…) a :

- des informations intrinsèques : forme, police, taille, etc.

- des informations extrinsèques : position (x, y), couleur locale, etc.

Composant du pattern	Exemple
`Flyweight`	Objet représentant la lettre “A” avec la police Arial.
`repeatingState`	“A”, Arial (identique pour toutes les lettres “A” dans le document).
`Context`	Chaque occurrence de “A” à une position donnée (x, y).
`uniqueState`	Coordonnées du caractère à l’écran.
`Client`	Editeur du texte.
`FlyweightFactory`	Gère un cache de lettres “A”, “B”, “C”, etc.

Exemple : Editeur de texte

```
import java.util.*;

// Flyweight
interface Glyph {
    void draw(int x, int y);
}

// Concrete Flyweight
class CharacterFlyweight implements Glyph {
    private final char symbol; // état intrinsèque partagé

    public CharacterFlyweight(char symbol) {
        this.symbol = symbol;
    }

    public void draw(int x, int y) { // état extrinsèque
        System.out.println("Caractère " + symbol + " à (" + x + "," + y + ")");
    }
}
```

```
// Factory
class GlyphFactory {
    private final Map<Character, Glyph> pool = new HashMap<>();

    public Glyph getCharacter(char c) {
        if (!pool.containsKey(c)) {
            pool.put(c, new CharacterFlyweight(c));
        }
        return pool.get(c);
    }
}
```

```
// Client
public class FlyweightDemo {
    public static void main(String[] args) {
        GlyphFactory factory = new GlyphFactory();
        String text = "ABABAC";
        int x = 0;
        for (char c : text.toCharArray()) {
            Glyph g = factory.getCharacter(c);
            g.draw(x++, 1);
        }
        System.out.println("Nombre d'objets créés : " + factory.pool.size());
    }
}
```

```
Caractère A à (0,1)
Caractère B à (1,1)
Caractère A à (2,1)
Caractère B à (3,1)
Caractère A à (4,1)
Caractère C à (5,1)
Nombre d'objets créés : 3
```

Exercice d’application

Problématique:

Dans un jeu vidéo ou une simulation 3D, on peut avoir des dizaines de milliers d’arbres.

Créer un objet complet pour chaque arbre (avec son image et sa texture) serait trop lourd en mémoire.

On applique le pattern Flyweight pour partager les données communes à tous les arbres d’une même espèce

Exercice d’application

Travail demandé:

Implémentez un système simple où :

- Chaque arbre a un type partagé (TreeType) : espèce, couleur, texture.
- Chaque arbre a aussi des caractéristiques uniques : position X/Y, taille.
- Une fabrique réutilise les TreeType déjà créés pour éviter les doublons.
- Le programme crée plusieurs arbres et affiche combien d’objets partagés ont été créés.

Exercice d’application

Exemple de résultat attendu:

Affichage arbre [Chêne] couleur=Vert, texture=Texture1 à (10,20), taille=5
Affichage arbre [Chêne] couleur=Vert, texture=Texture1 à (15,25), taille=6
Affichage arbre [Pin] couleur=Vert Foncé, texture=Texture2 à (50,60), taille=8
Affichage arbre [Chêne] couleur=Vert, texture=Texture1 à (70,80), taille=10

Nombre de TreeType créés : 2