المدرسة الوطنية العليا للذكاء الاصطناعي وعلوم المعطيات - تارودانت
ÉCOLE NATIONALE SUPÉRIEURE DE L'INTELLIGENCE ARTIFICIELLE
ET SCIENCES DES DONNÉES - TAROUDANT

Compte Rendu TD4

Patterns de Conception - Exercice 2

Module : Génie Logiciel et Gestion de Projet

Filière : MGSI - Semestre 7

Réalisé par : ESSALEHY Badr-Eddine

Professeur : Pr. SAAD Basma

Année universitaire : 2025-2026

Table des matières

1 Introduction

Ce compte rendu présente la résolution de l'exercice 2 du TD4 qui traite de la refonte d'un jeu éducatif pour enfants. L'objectif principal est d'améliorer l'architecture logicielle en introduisant une interface commune tout en réutilisant le code existant grâce au patron de conception **Adapter**.

Le problème initial réside dans la spécificité des classes existantes (`LeChat`, `LaVache`) qui ne permettent pas une manipulation polymorphique des animaux. La solution proposée utilise le pattern Adapter pour créer un pont entre l'ancienne architecture et la nouvelle, garantissant ainsi la réutilisabilité du code sans réécriture.

2 Problématique

2.1 Version initiale du système

La première version du jeu modélise chaque animal avec des méthodes spécifiques :
— `LeChat` : possède les méthodes `formeChat()` et `criChat()`
— `LaVache` : possède les méthodes `criVache()` et `formeVache()`

2.2 Limitations identifiées

Cette approche présente plusieurs inconvénients majeurs :
— **Absence de polymorphisme** : impossible de manipuler les animaux via une interface commune
— **Manque de flexibilité** : l'ajout de nouveaux animaux nécessite des modifications dans le code client
— **Code non réutilisable** : difficile de créer des collections hétérogènes d'animaux
— **Maintenance complexe** : chaque animal nécessite une gestion spécifique

3 Solution proposée

3.1 Question 1 : Nouvelle architecture avec interface commune

3.1.1 Conception

La nouvelle architecture repose sur une interface `Animal` qui définit un contrat commun pour tous les animaux. Cette approche permet d'exploiter le polymorphisme et facilite l'extension du système.

Diagramme de classes :

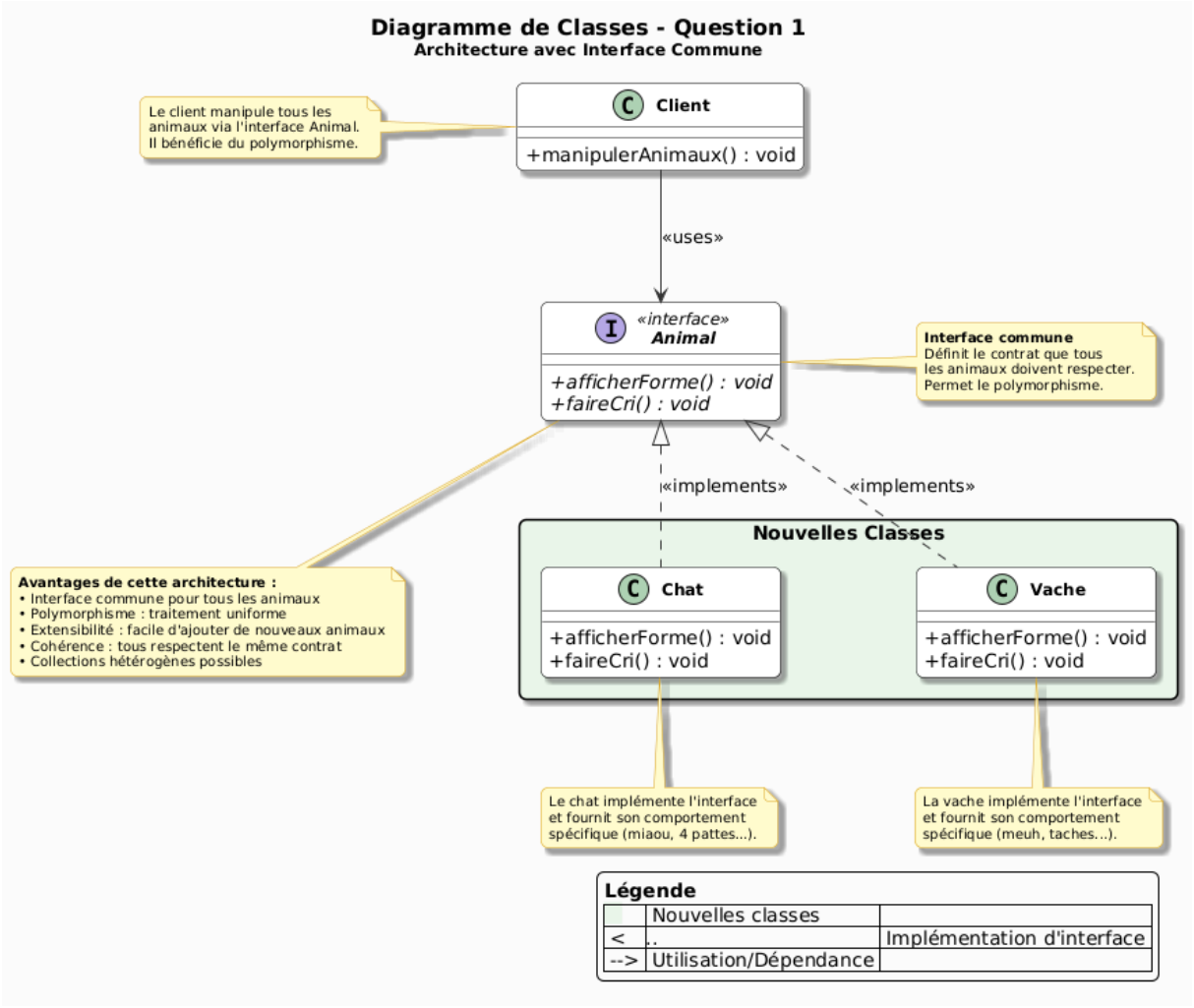

FIGURE 1 – Architecture de la nouvelle version avec interface Animal

3.1.2 Interface Animal

L'interface `Animal` définit deux méthodes abstraites que chaque animal doit implémenter :

FIGURE 2 – Code de l'interface Animal

3.1.3 Implémentation : Classe Chat

La classe `Chat` implémente l'interface `Animal` et fournit son propre comportement :

FIGURE 3 – Implémentation de la classe Chat

3.1.4 Implémentation : Classe Vache

De manière similaire, la classe `Vache` implémente l'interface avec ses spécificités :

FIGURE 4 – Implémentation de la classe Vache

3.1.5 Avantages de cette approche

— **Polymorphisme** : possibilité de manipuler tous les animaux via l'interface `Animal`
— **Extensibilité** : ajout facile de nouveaux animaux sans modifier le code existant
— **Cohérence** : toutes les classes d'animaux respectent le même contrat
— **Collections hétérogènes** : création possible de listes d'animaux variés

3.2 Question 2 : Réutilisation du code avec le Pattern Adapter

3.2.1 Principe du Pattern Adapter

Le pattern Adapter permet de faire collaborer des classes dont les interfaces sont incompatibles. Il agit comme un traducteur entre l'ancienne interface et la nouvelle, permettant ainsi de réutiliser le code existant sans modification.

Rôles dans notre contexte :

— **Target** : l'interface `Animal` (nouvelle architecture)
— **Adaptee** : les anciennes classes `LeChat` et `LaVache`
— **Adapter** : les classes `ChatAdapter` et `VacheAdapter`
— **Client** : le programme principal qui utilise l'interface `Animal`

3.2.2 Diagramme de classes avec Adapter

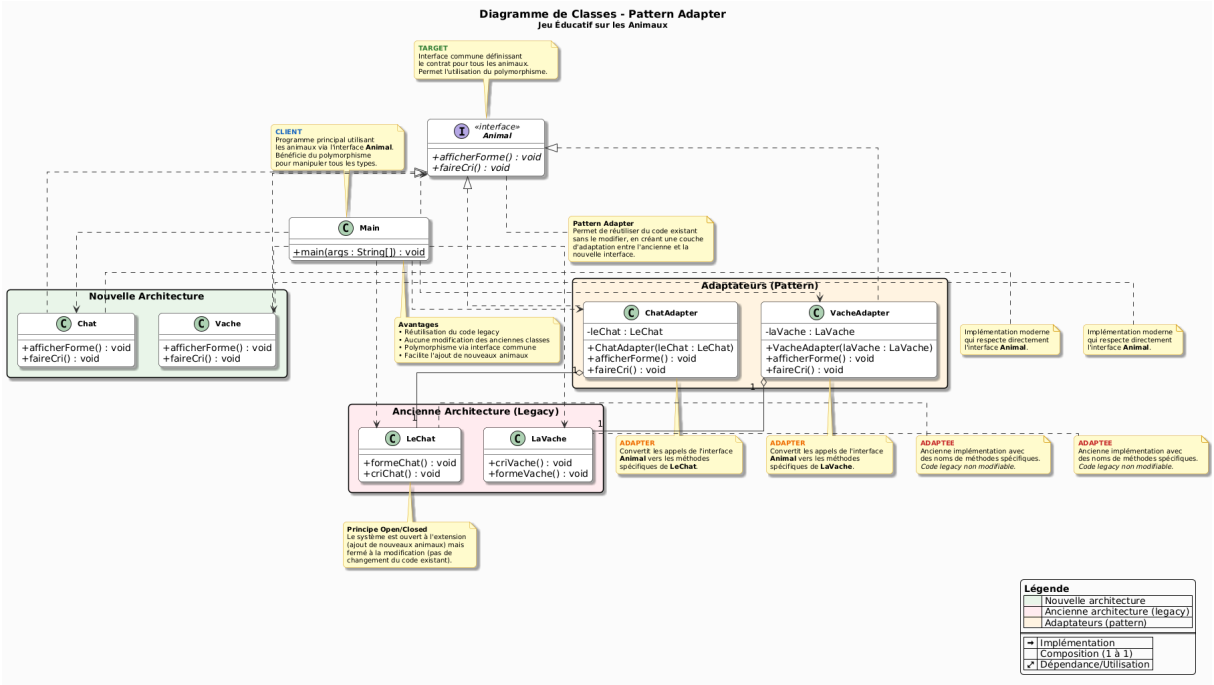

FIGURE 5 – Architecture complète avec le Pattern Adapter

3.2.3 Classes anciennes (Adaptees)

Les classes existantes restent inchangées pour garantir la compatibilité avec l'ancien système.

Classe LeChat :

LeChat.java

```java
public class LeChat {  no usages
    public void formeChat() {  no usages
        System.out.println("[Ancienne version] Forme du chat : petit félin domestique");
    }

    public void criChat() {  no usages
        System.out.println("[Ancienne version] Cri du chat : Miaou miaou!");
    }

}
```

FIGURE 6 – Classe LeChat (version originale)

Classe LaVache :

LaVache.java

```java
public class LaVache {  no usages

    public void criVache() {  no usages
        System.out.println("[Ancienne version] Cri de la vache : Meuh meuh!");
    }

    public void formeVache() {  no usages
        System.out.println("[Ancienne version] Forme de la vache : gros bovidé");
    }
}
```

FIGURE 7 – Classe LaVache (version originale)

3.2.4 Adaptateurs

Les adaptateurs font le pont entre les anciennes classes et la nouvelle interface.
ChatAdapter :

ChatAdapter.java

```java
public class ChatAdapter implements Animal{
    private LeChat leChat;  3 usages

    public ChatAdapter(LeChat leChat) {  no
        this.leChat = leChat;
    }

    @Override  no usages
    public void afficherForme() {
        leChat.formeChat();
    }

    @Override  no usages
    public void faireCri() {
        leChat.criChat();
    }
}
```

FIGURE 8 – Adaptateur pour la classe LeChat

VacheAdapter :

VacheAdapter.java

```java
public class VacheAdapter implements Animal{

    private LaVache laVache;  3 usages

    public VacheAdapter(LaVache laVache) {
        this.laVache = laVache;
    }

    @Override  no usages
    public void afficherForme() {
        laVache.formeVache();
    }

    @Override  no usages
    public void faireCri() {
        laVache.criVache();
    }
}
```

FIGURE 9 – Adaptateur pour la classe LaVache

3.2.5 Programme de démonstration

Le programme principal illustre l'utilisation du polymorphisme et de l'adaptateur :

Main.java

```java
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        System.out.println("=== Démonstration du Pattern Adapter ===\n");

        System.out.println("--- 1. Utilisation des nouvelles classes ---");
        Animal chat = new Chat();
        Animal vache = new Vache();

        chat.afficherForme();
        chat.faireCri();
        System.out.println();

        vache.afficherForme();
        vache.faireCri();
        System.out.println();

        System.out.println("--- 2. Utilisation des anciennes classes via Adapter ---");
        LeChat ancienChat = new LeChat();
        LaVache ancienneVache = new LaVache();

        Animal chatAdapte = new ChatAdapter(ancienChat);
        Animal vacheAdaptee = new VacheAdapter(ancienneVache);

        chatAdapte.afficherForme();
        chatAdapte.faireCri();
        System.out.println();

        vacheAdaptee.afficherForme();
        vacheAdaptee.faireCri();
        System.out.println();

        System.out.println("--- 3. Collection hétérogène d'animaux (polymorphisme) ---");
        List<Animal> troupeau = new ArrayList<>();
        troupeau.add(chat);
        troupeau.add(vache);
        troupeau.add(chatAdapte);
        troupeau.add(vacheAdaptee);

        System.out.println("Concert des animaux :");
        for (Animal animal : troupeau) {
            animal.faireCri();
        }
    }
}
```

FIGURE 10 – Programme de test et démonstration

4 Analyse des résultats

4.1 Exécution du programme

L'exécution du programme produit la sortie suivante :

```
=== Démonstration du Pattern Adapter ===

--- 1. Utilisation des nouvelles classes ---
Le chat a quatre pattes, des moustaches et une queue.
Le chat fait : Miaou!

La vache est grande, avec des taches noires et blanches.
La vache fait : Meuh!

--- 2. Utilisation des anciennes classes via Adapter ---
[Ancienne version] Forme du chat : petit félin domestique
[Ancienne version] Cri du chat : Miaou miaou!

[Ancienne version] Forme de la vache : gros bovidé
[Ancienne version] Cri de la vache : Meuh meuh!

--- 3. Collection hétérogène d'animaux (polymorphisme) ---
Concert des animaux :
Le chat fait : Miaou!
La vache fait : Meuh!
[Ancienne version] Cri du chat : Miaou miaou!
[Ancienne version] Cri de la vache : Meuh meuh!
```

FIGURE 11 – Résultat de l'exécution du programme

4.2 Avantages de la solution avec Adapter

1. **Réutilisation totale du code** : aucune ligne de code des anciennes classes n'a été modifiée
2. **Compatibilité ascendante** : l'ancien système continue de fonctionner indépendamment
3. **Flexibilité** : possibilité de mélanger anciennes et nouvelles implémentations
4. **Séparation des responsabilités** : l'adaptateur gère uniquement la conversion d'interface
5. **Facilite la migration progressive** : permet de passer graduellement à la nouvelle architecture

4.3 Impact d'un nouvel animal

L'ajout d'un nouvel animal (par exemple, un chien) se fait simplement :

— **Option 1** : créer une nouvelle classe `Chien` implémentant `Animal`

— **Option 2** : si une classe `LeChien` existe, créer un `ChienAdapter`

Aucune modification du code existant n'est nécessaire, respectant ainsi le principe Open/Closed.

5 Conclusion

Cette étude de cas démontre l'efficacité du pattern Adapter pour moderniser un système existant tout en préservant les investissements logiciels antérieurs. La solution proposée offre :

— Une architecture propre basée sur une interface commune

— La réutilisation complète du code legacy sans modification

— Une extensibilité maximale pour les évolutions futures

— Un respect des principes SOLID, notamment le principe Open/Closed

Le pattern Adapter s'avère particulièrement pertinent dans les contextes de refactoring et de migration progressive, permettant de concilier innovation architecturale et contraintes de compatibilité.