Conception et Programmation objet Avancées

Sylvie Coste

IUT de Lens

2021–2022

Plan

Étude d'un cas concret (1)

Un problème de canard ...

On veut concevoir un simulateur de mare aux canards : le simulateur gère différents canards, il peut les faire cancaner et nager dans la mare. Parmi les premiers canards, on considère le colvert, le mandarin et le canard en plastique.

- Concevez un diagramme de classe pour représenter ce simulateur

Étude d'un cas concret (2)

... qui se mettent à voler (ou pas) !

On souhaite maintenant ajouter une méthode permettant de faire voler les canards et un canard télécommandé (il vole avec un moteur et ne cancane pas).

- Où allez-vous définir la méthode *voler()* ?
- Est-ce que cela vous paraît souhaitable ?
- Quels sont les inconvénients ?

Diagramme de classes : simulateur

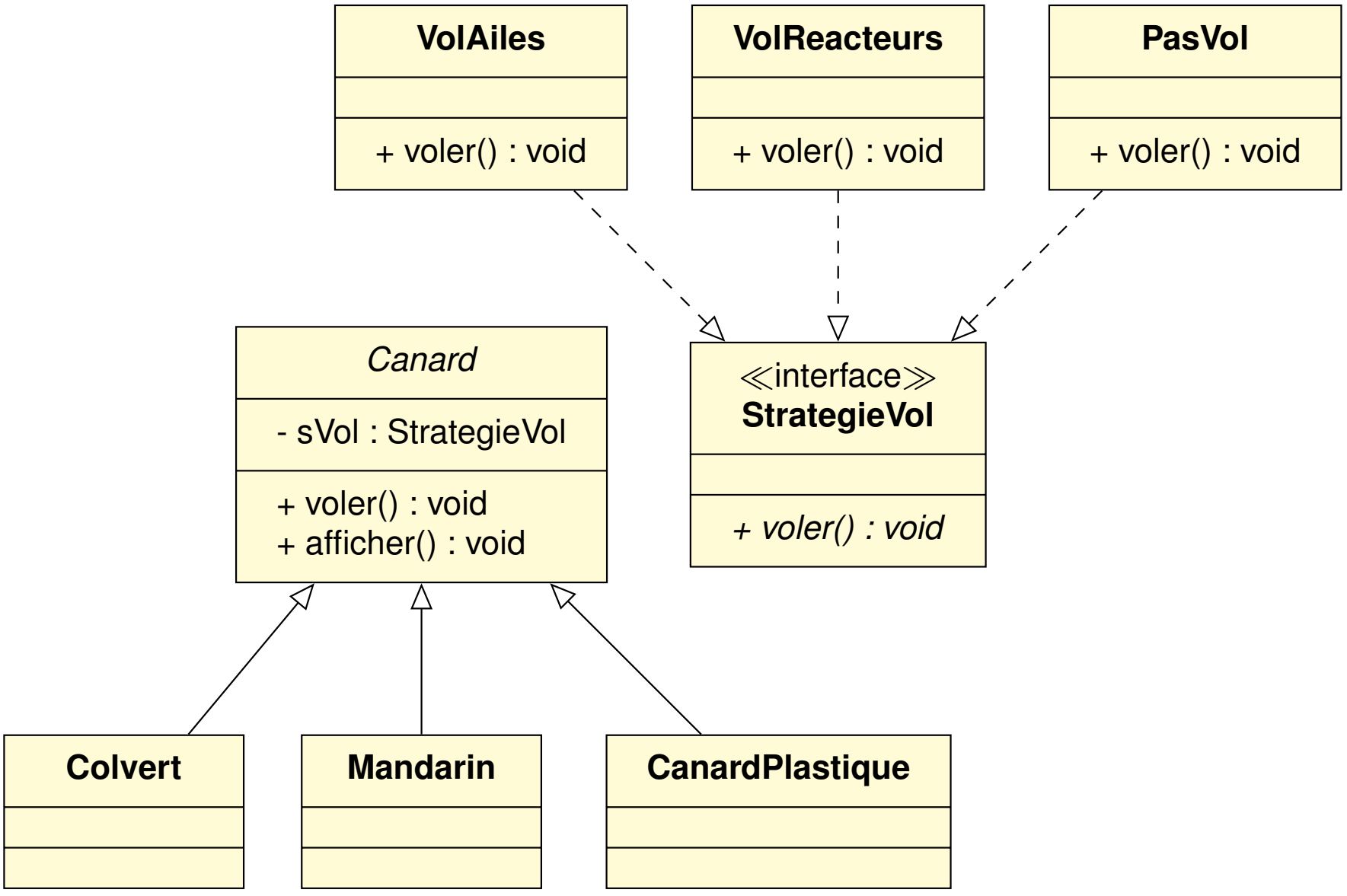
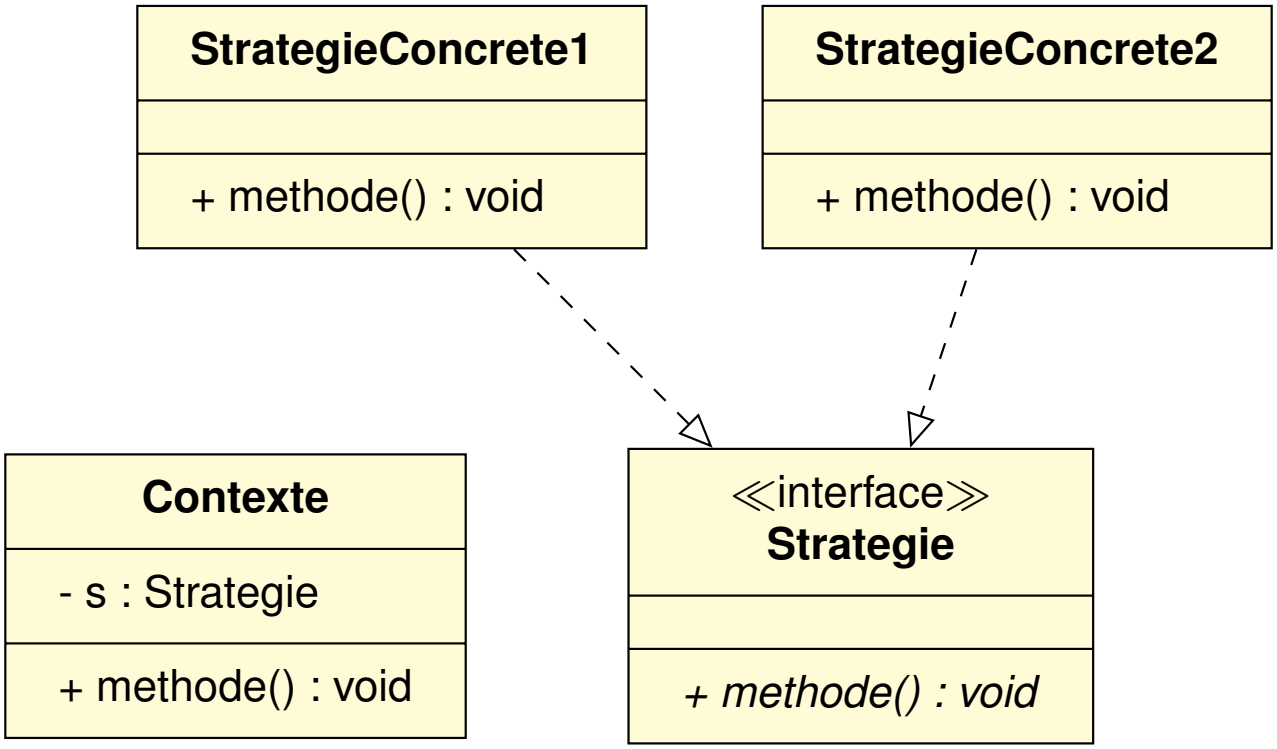


Diagramme de classes : cas général



Design Pattern Stratégie

Objectif

Permettre d'utiliser différents algorithmes identiques sur le plan conceptuel en fonction du contexte

Problème

La sélection de l'algorithme dépend du client à l'origine de la demande ou des données à traiter

Solution

Séparer la sélection de l'algorithme de son implantation

Conséquence

Le patron Stratégie définit une famille d'algorithmes

Les constituants du DP stratégie

Stratégie

- Définit une interface commune à tous les algorithmes représentés

StratégieConcrète

- Implémente l'algorithme

Contexte

- Est composé à l'aide d'un objet Stratégie Concrète
- Gère une référence à un objet Stratégie
- Peut définir une interface qui permet à Stratégie d'accéder à ses données

Indications d'utilisation du DP stratégie

- Plusieurs classes apparentées ne diffèrent que par leur comportement : les stratégies permettent d'apparier une classe avec un comportement
- Diverses variantes d'un algorithme coexistent (par exemple, on définit des algorithmes avec différents compromis temps d'exécution/encombrement mémoire) : les stratégies permettent de sélectionner la variante
- Un algorithme utilise des données que les clients n'ont pas à connaître : la stratégie permet de masquer les structures internes spécifiques à un algorithme
- Une classe définit de nombreux comportements qui figurent sous forme de conditionnelles multiples : on déplace les comportements dans des classes stratégies qui permettent de supprimer ces conditionnelles

Le code du design pattern stratégie (1)

L'interface *StrategieVol*

```
public interface StrategieVol{
  void voler();
}
```

La classe *VolAvecAiles*

```
public class VolAvecAiles implements StrategieVol{
  public void voler(){
    System.out.println("Je vole !");
  }
}
```

Le code du design pattern stratégie (2)

La classe abstraite *Canard*

```
public abstract class Canard {
  private StrategieVol strategieVol;

  public Canard(StrategieVol strategieVol) {
    this.strategieVol = strategieVol;
  }

  public void setStrategieVol (StrategieVol strategieVol) {
    this.strategieVol = strategieVol;
  }

  public void voler() {
    strategieVol.voler();
  }

  public abstract void afficher();
}
```

Le code du design pattern stratégie (3)

La classe *Colvert*

```
public class Colvert extends Canard {
  public Colvert() {
    super(new VolerAvecAiles());
  }

  public void afficher(){
    System.out.println("Je suis un vrai beau colvert");
  }
}
```

Plan

Stratégie

Adaptateur

Étude du problème (1)

Encore un problème de canard

On reprend notre simulateur de mare aux canards. La mare est un lieu où les canards viennent cancaner et voler.
Tous les canards doivent implémenter les deux méthodes *cancaner()* et *voler()*.

- Quelle modélisation proposez-vous ?

Étude du problème (2)

Arrivée des dindons

Des dindons sauvages viennent fréquenter votre mare. Ils volètent par intermittence et ne cancanent pas mais glougloutent. Vous devez cependant trouver une solution pour qu'ils cancanent et volent dans votre simulateur !

- Comment modifier le diagramme de classes pour résoudre ce problème ?

Diagramme de classes : la mare

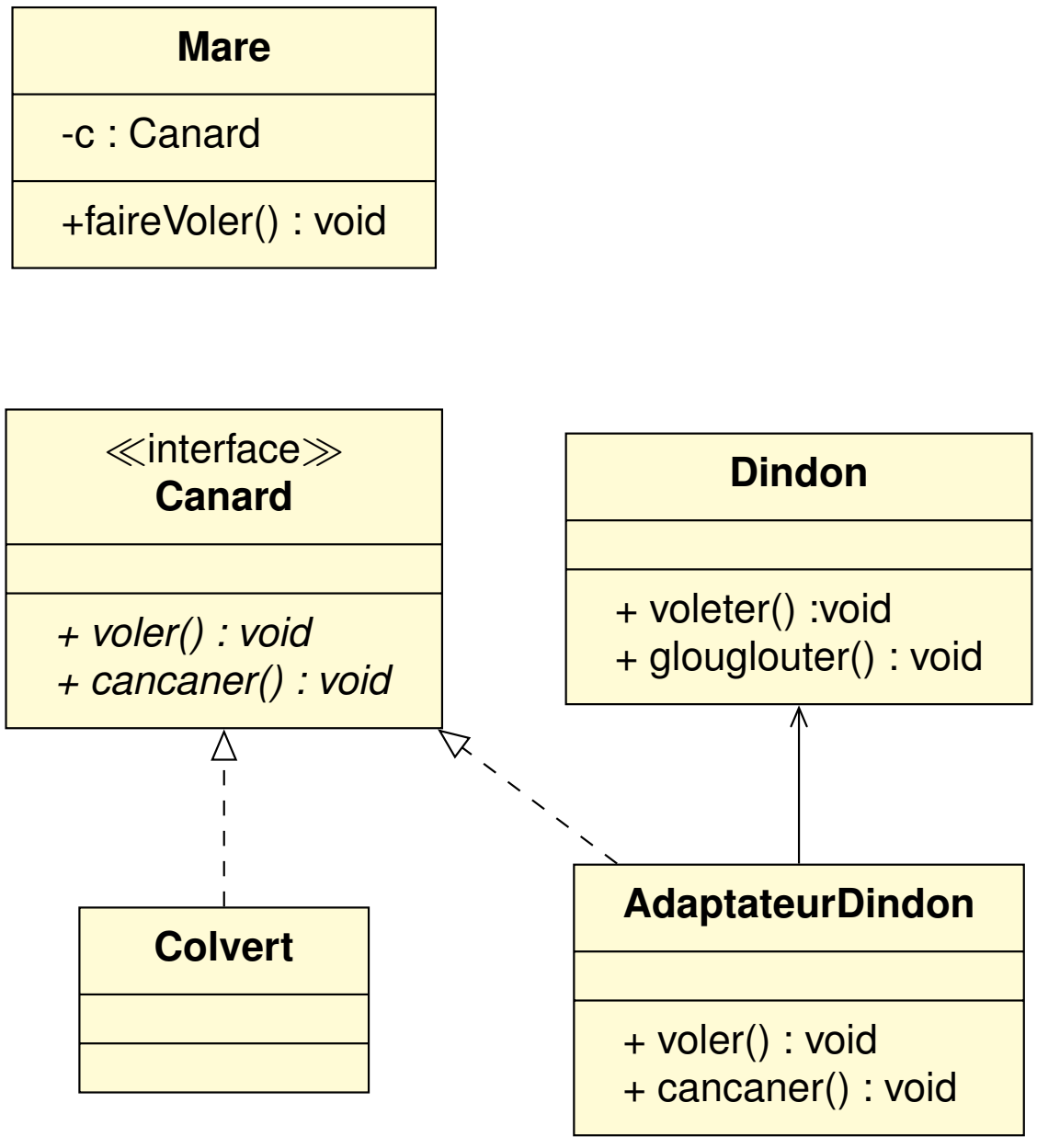
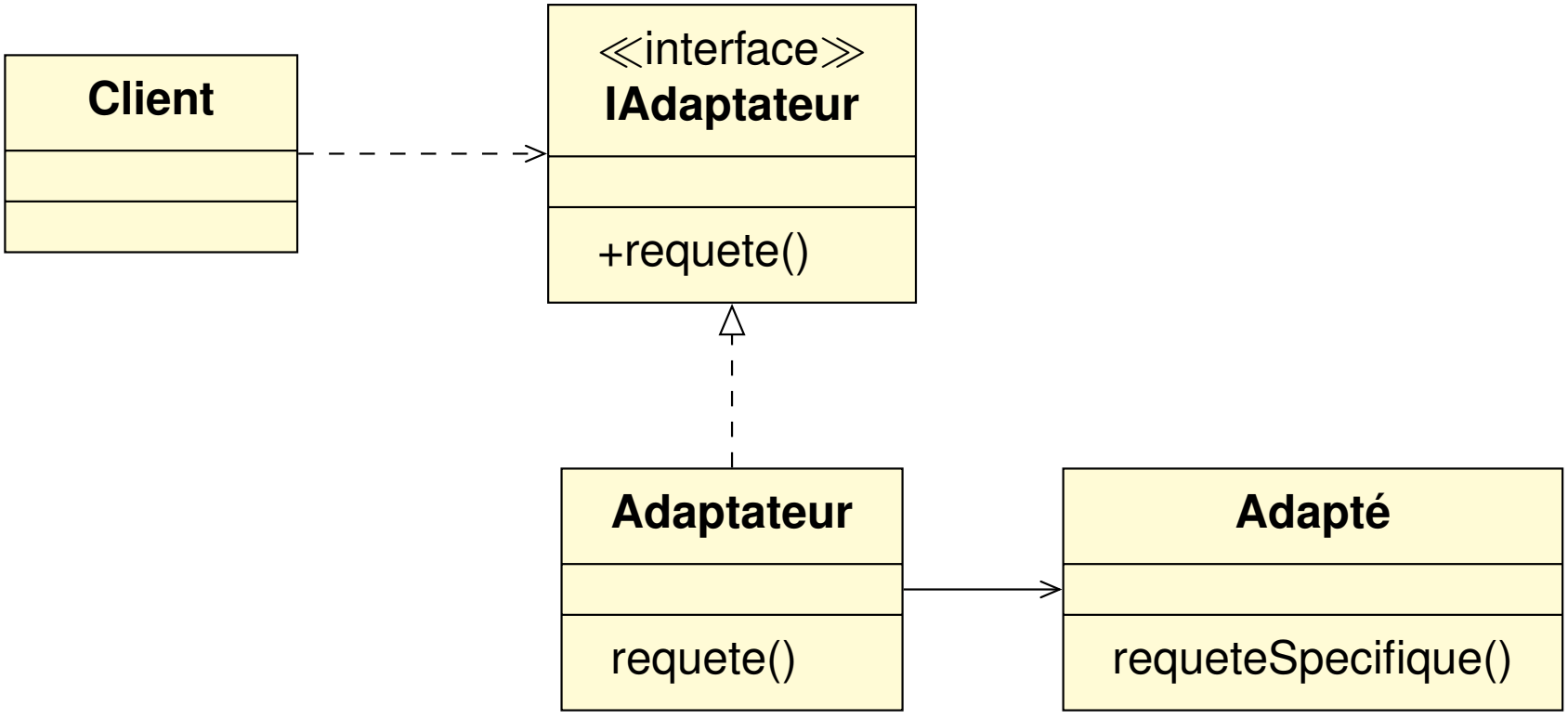


Diagramme de classe : cas général



Design pattern adaptateur

Objectif

- Faire correspondre à une interface donnée un objet (ou une classe) que l'on ne contrôle pas

Problème

- Un système a les bonnes données et les bonnes méthodes, mais pas les bonnes interfaces

Solution

- Délégation (objet) ou héritage (adaptateur de classe)

Conséquence

- Des objets existants peuvent être intégrés à de nouvelles structures de classes sans être limités par leur interface

Les constituants du DP adaptateur

IAdaptateur

- Définit les méthodes requises

Adapté

- Fournit les méthodes mais n'implémente pas l'interface

Adaptateur

- Adapte les méthodes de la classe `Adapté` pour qu'elles correspondent à la cible de l'`IAdaptateur`

Indications d'utilisation du DP adaptateur

- Une API (interface de programmation applicative) développée par un tiers contient le code recherché mais la signature des méthodes ne correspond pas
- D'anciennes classes doivent être utilisées sans réécrire tout leur code

Le code de l'adaptateur

La classe AdaptateurDindon

```
public class AdaptateurDindon implements Canard {
  private Dindon dindon;

  public AdaptateurDindon() {
          this.dindon = new Dindon();
  }

  public void cancaner() {
    dindon.glouglouter();
  }

  public void voler() {
    for(int i=0; i < 5; i++) {
      dindon.voleter();
    }
  }
}
```

Le code de l'adaptateur

Une partie du code de la Mare

```
public class Mare {
  public static void main(String[] args) {
    Canard canard = new Colvert();
    Canard adaptateurDindon = new AdaptateurDindon();

    System.out.println("\nCanard ");
    canard.cancaner();
    canard.voler();

    System.out.println("\nAdaptateurDindon ");
    adaptateurDindon.cancaner();
    adaptateurDindon.voler();
  }
}
```