Cours M3105 : Conception et programmation objet avancées

L'exemple présenté dans ce cours est tiré du livre Design patterns – Tête la première, E. E. Freeman, ed. O'Reilly

IUT Villetaneuse

semaine du 7 novembre 2016

Motivation Une bonne conception Le design Pattern Stratégie

Plan

Gérer des comportements

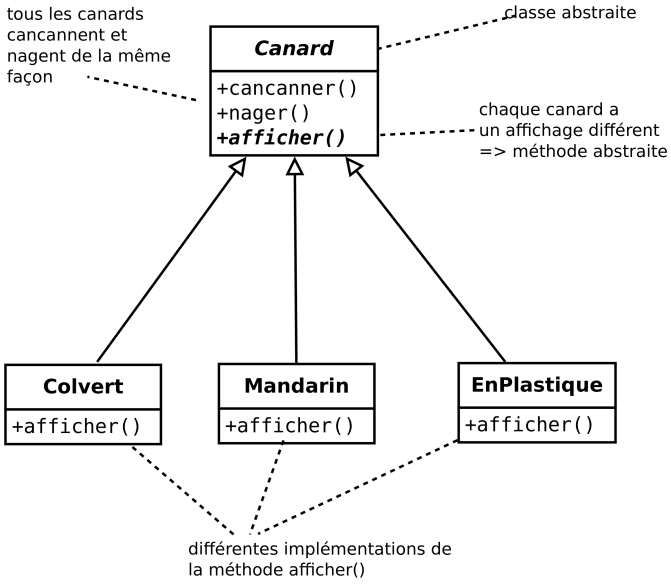
Le comportement d'une classe X doit être

- Extensible
  - On peut ajouter un comportement
- Modifiable
  - On peut changer un comportement par un autre
- Dynamique
  - On peut modifier le comportement en cours d'exécution

.. sans modifier les classes qui utilisent X

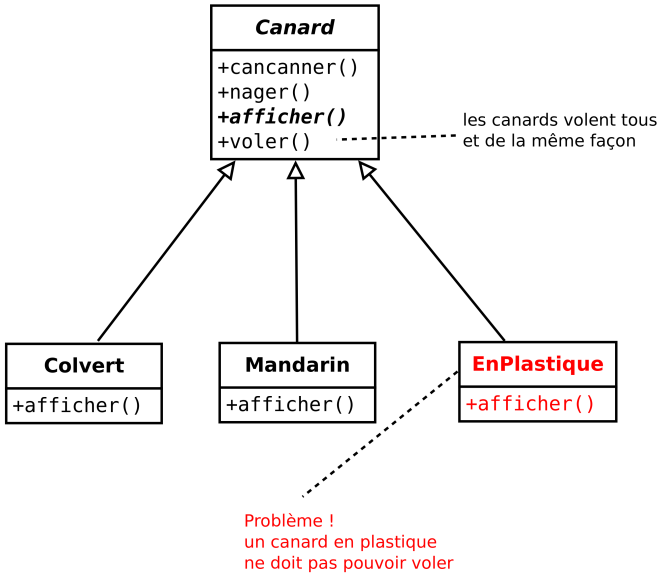
Représenter des comportements

Représenter des races de canards et leurs comportements



Ajouter un comportement pose problème

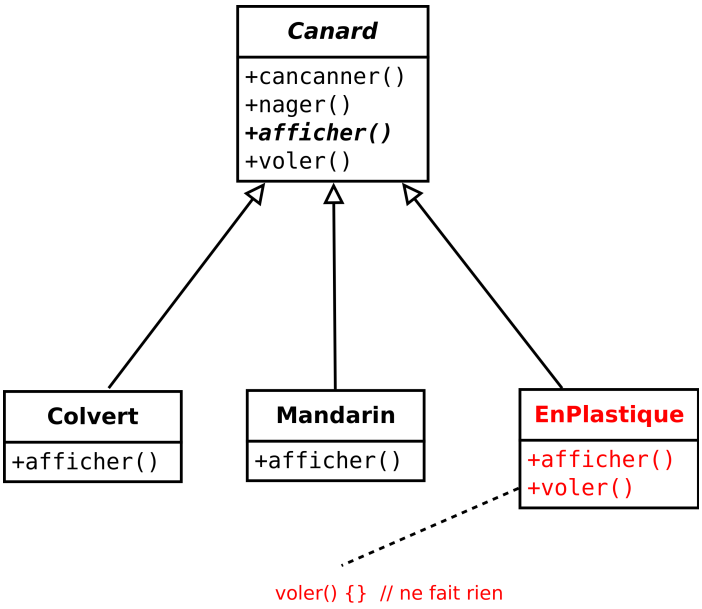
Désormais les canards volent



Ajouter un comportement à la classe Canard l'impose à toute sa descendance

Une mauvaise solution

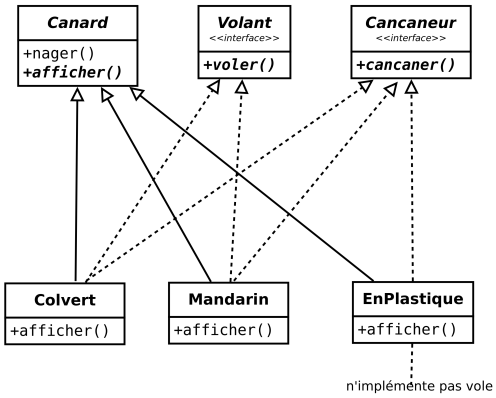
Redéfinir la méthode voler() dans la classe EnPlastique



Restrictif : il faut prévoir différentes façons de voler selon les classes

Une autre mauvaise solution

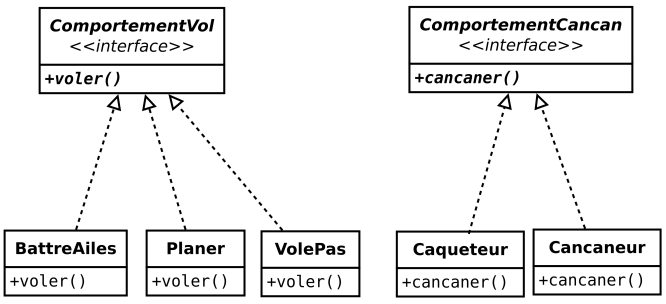
Créer des interfaces pour que les canards puissent voler et cancaner de différentes façons, et que d'autres ne volent pas ou ne cancanent pas



un canard EnPlastique ne vole plus mais ! chaque classe qui implémente Volant et Cancaneur doit fournir un code pour les méthodes voler et cancaner. On a perdu la factorisation du code.

Séparer ce qui peut changer du reste

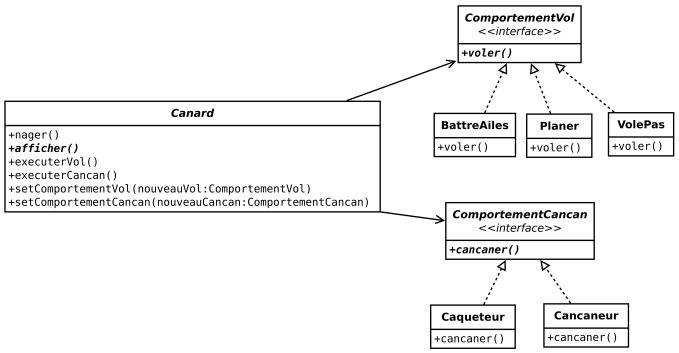
Identifier les comportements susceptibles d'être modifiés ou qui varient d'une classe à l'autre (cancaner, voler) et les encapsuler



Chaque comportement réside de façon abstraite dans une interface, et de façon concrète dans ses implémentations

Intégrer les comportements isolés

La classe Canard est associée à un ComportementVol et un ComportementCancan



Délégation des comportements

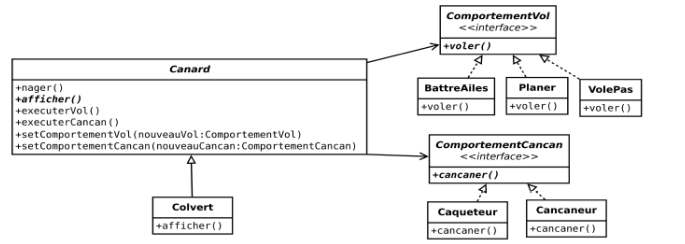
La classe Canard délègue les comportements encapsulés

```
public class Canard
{
    private ComportementVol unComportementVol ;
    private ComportementCancan unComportementCancan ;

    public void executerVol()
    {
        this.unComportementVol.voler() ;
    }

    public void executerCancan()
    {
        this.unComportementCancan.cancaner() ;
    }

}
```

La classe *Canard* délègue la gestion du vol à l'objet *unComportementVol*

..et la gestion du cancanement à l'objet *unComportementCancan*

Initialiser les variables de comportement




```
public class Colvert extends Canard
{
    public Colvert()
    {
        this.SetComportementVol (new BattreAiles()) ;
        this.SetComportementCancan (new Caqueteur()) ;
    }
}
```

On affecte les comportements désirés au *Colvert* qui vient d'être créé

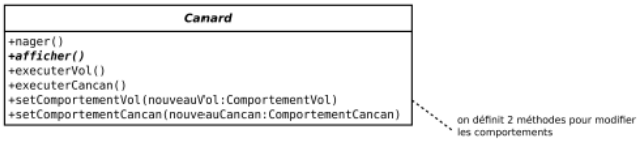
Tester les comportements

```
public class Canard
{
    private ComportementVol unComportementVol ;
    private ComportementCancan unComportementCancan ;

    public void executerVol()
    {this.unComportementVol.voler();}

    public void executerCancan()
    {this.unComportementCancan.cancaner();}
}
```

```
public class Colvert
             extends Canard
{
    public Colvert()
    {

this.SetComportementVol(new
BattreAiles()) ;

this.SetComportementCancan (new
Caqueteur()) ;
    }
}
```

```
public class TestCanard
{
    public static void
            main(String[] args)
    {
        Canard c = new Colvert() ;

        c.executerCancan();
        c.executerVol() ;
    }
}
```

Modification dynamique du comportement




```
public class Canard
{
    ...
    public void setComportementVol(ComportementVol nouveauVol)
    {
        this.unComportementVol = nouveauVol ;
    }

    public void setComportementCancan(ComportementCancan nouveauCancan)
    {
        this.unComportementCancan = nouveauCancan ;
    }
}
```


Tester les comportements dynamiques

```
public class TestCanard
{
    public static void
            main(String[] args)
    {
        Canard c   =  new Colvert() ;

        c.executerVol();
        c.executerCancan() ;

        c.setComportementVol(new Planer()) ;
        c.setComportementCancan(new Caqueter()) ;

        c.executerVol() ;
        c.executerCancan() ;
    }
}
```

« je bats des ailes »
« je cancane »

on change les comportements

« je plane »
« je caquette »

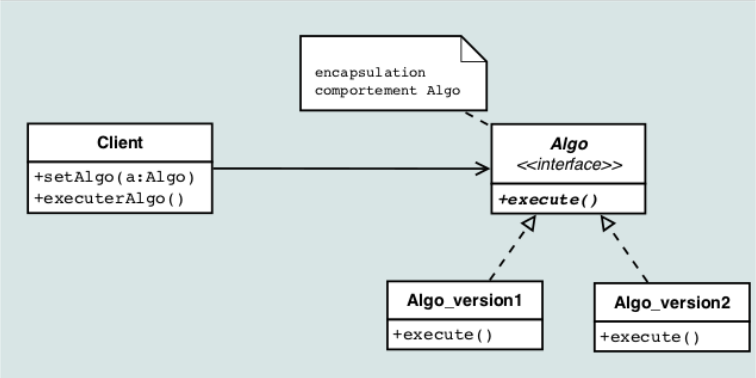
Le Design Pattern Stratégie

Définition

Le design pattern *Stratégie* définit une famille d'algorithmes (comportements), encapsule chacun d'eux et les rend interchangeables.
Il permet aux algorithmes de varier indépendamment des clients qui les utilisent.

Structure du design pattern Stratégie

Un client utilise un (des) algorithme(s) encapsulé(s)



Le client choisit, quand il le veut, la version de l'algorithme qui convient

Principes de conception rencontrés

Principes Généraux mis en oeuvre

- Identifier dans l'application ce qui peut varier et l'encapsuler dans des interfaces (et leurs implémentations)
  - On pourra modifier facilement les parties changeantes sans modifier les utilisateurs de ces parties
- Préférer la composition à l'héritage
  - L'héritage manque de souplesse et impose ce qui se définit dans la classe mère à sa dépendance
  - La composition permet de séparer les comportements