Objet et développement d'applications
Cours 2 - Pattern Strategy

Patrons de conception pour POO

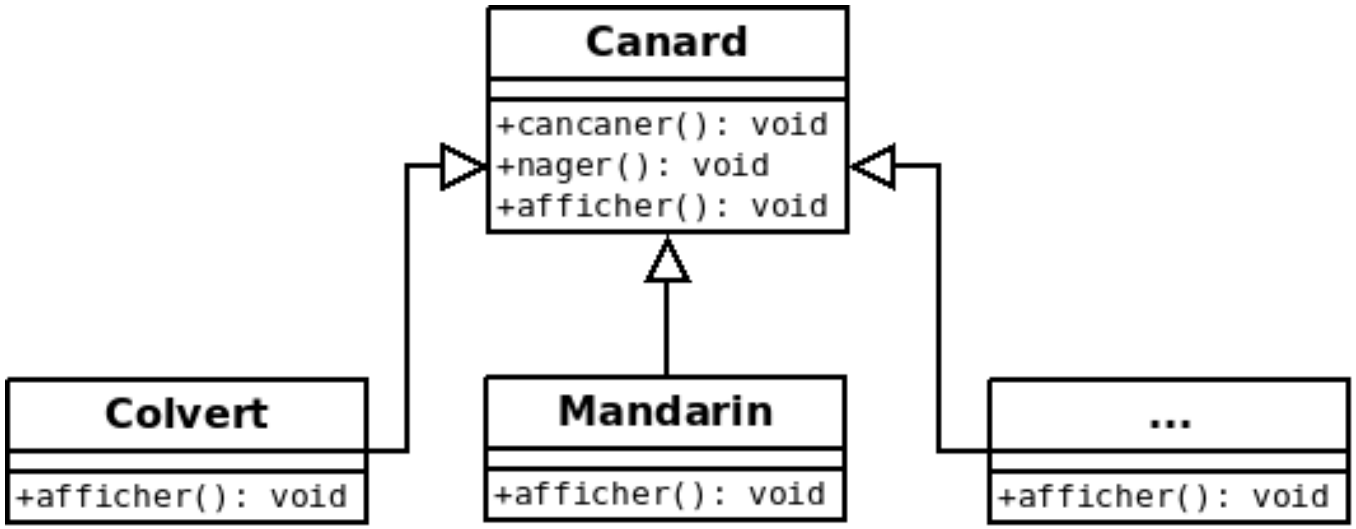
Florian Richoux

2014-2015

Une histoire de canards. . .

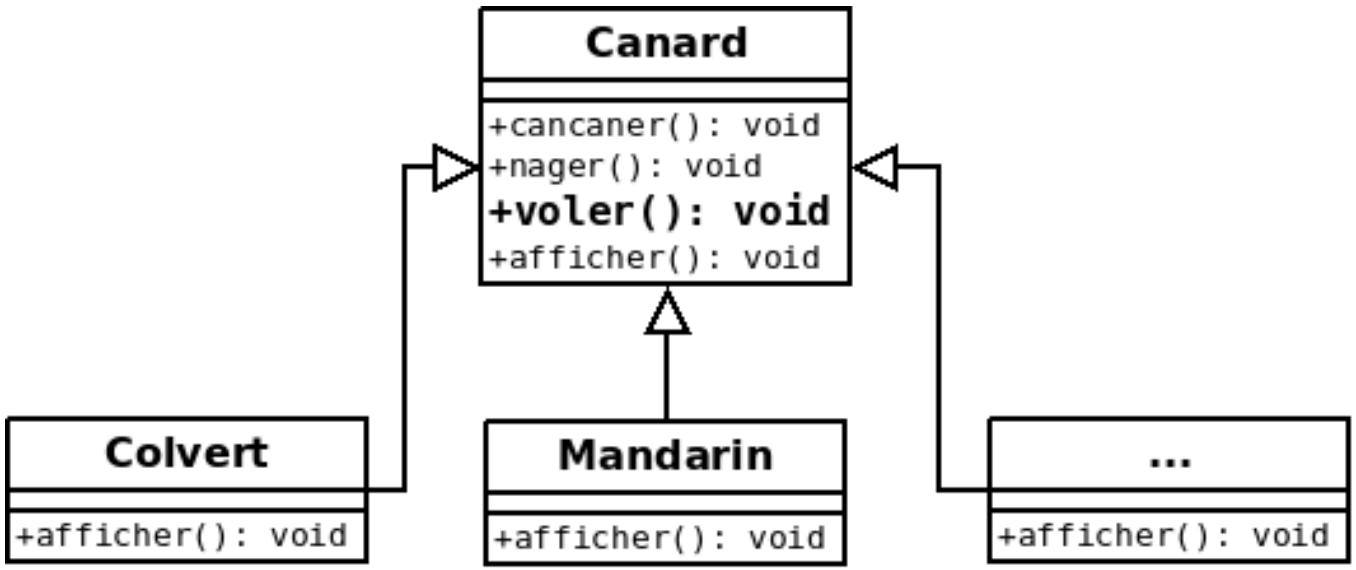
Faire des canards

Imaginez que l'on souhaite programmer le comportement de toutes sortes de canards.



Ajout d'une méthode

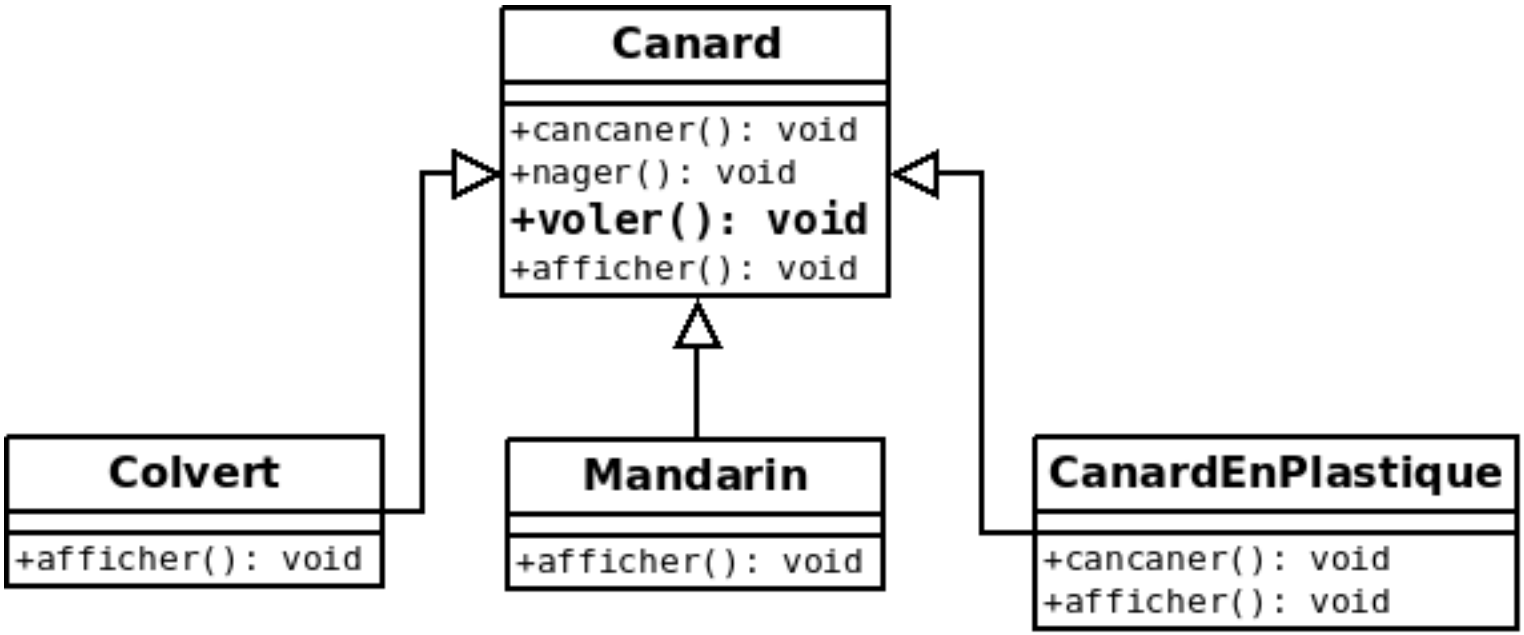
Comme tout va bien jusque là, on étends les fonctionnalités.



Problème après l'ajout d'une méthode

Pas de bol

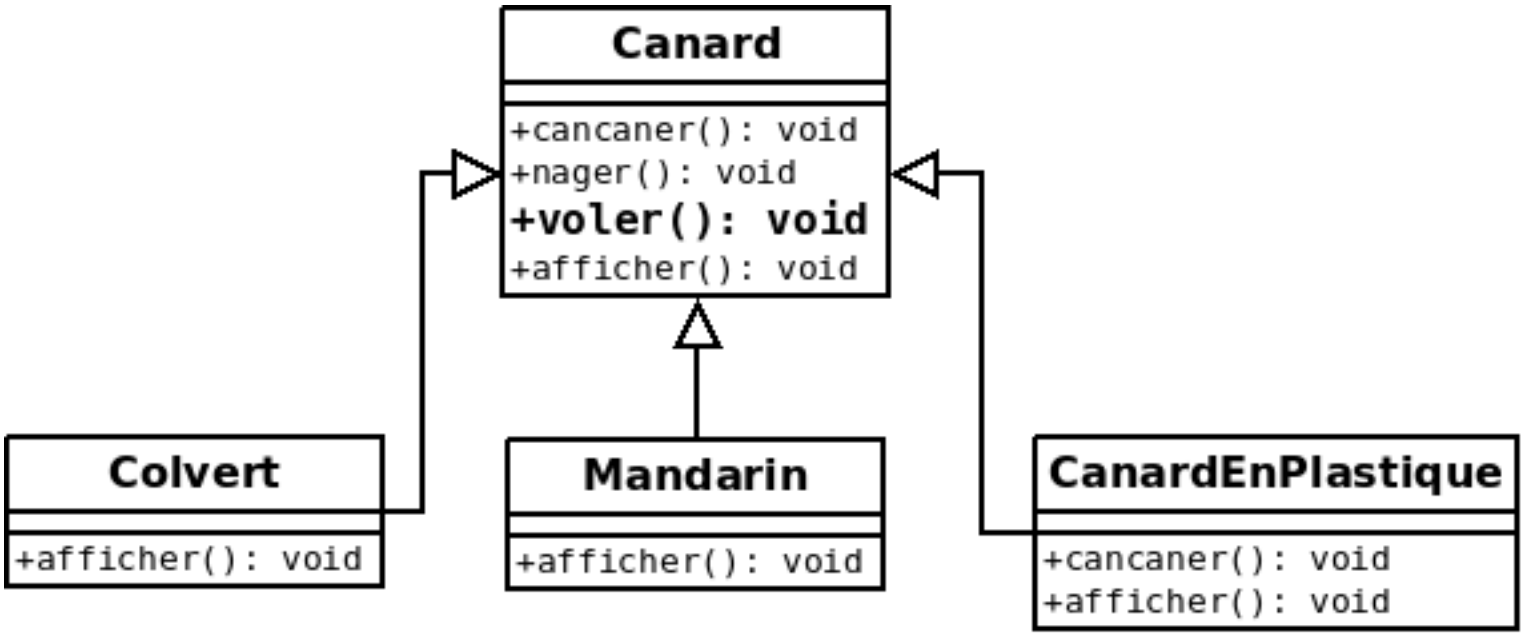
On a un canard en plastique, et ça vole pas...



Problème après l'ajout d'une méthode

Tiens tiens

CanardEnPlastique redéfinit la méthode *cancaner()* (pour couiner, sans doute). On pourrait pas redéfinir *voler()* et la laisser vide ?



Tentative de patch

Redéfinition de *voler()*

```
class CanardEnPlastique extends Canard
{
  public void cancaner()
  {
    System.out.println("Couinement");
  }

  public void afficher()
  {
    ...
  }

  public void voler() {}
}
```

Tentative de patch

On a résolu notre problème, mais...

Tentative de patch

On a résolu notre problème, mais...

Pas assez malin

En plus d'être inélégant, cette solution n'en est pas une :

- il faut modifier chaque classe qui posent problème.
- si un jour on rajoute d'autres méthodes à Canard, il faudra sans doute recommencer !

Tentative de patch

On a résolu notre problème, mais...

Pas assez malin

En plus d'être inélégant, cette solution n'en est pas une :

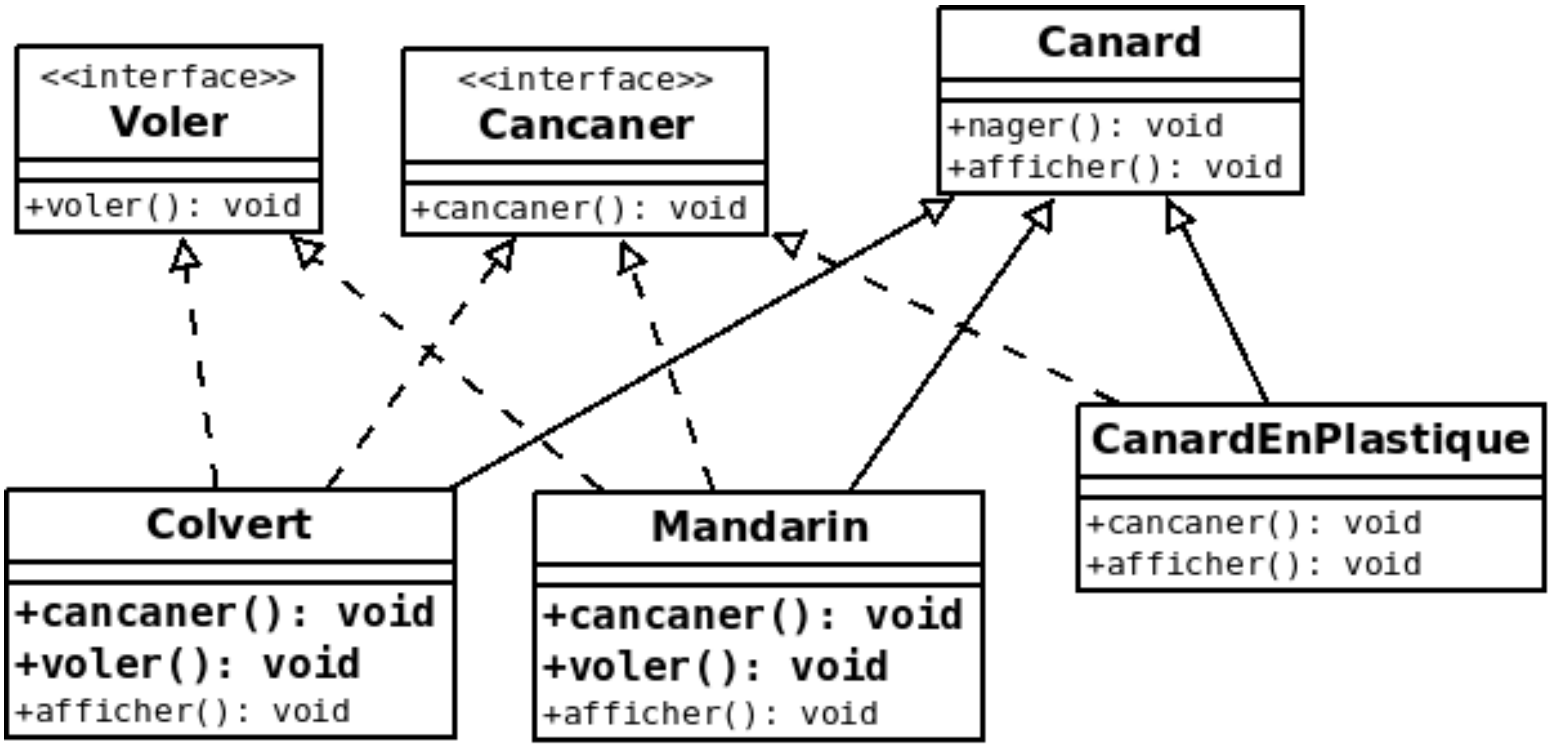
- il faut modifier chaque classe qui posent problème.
- si un jour on rajoute d'autres méthodes à Canard, il faudra sans doute recommencer !

Mauvaise structure

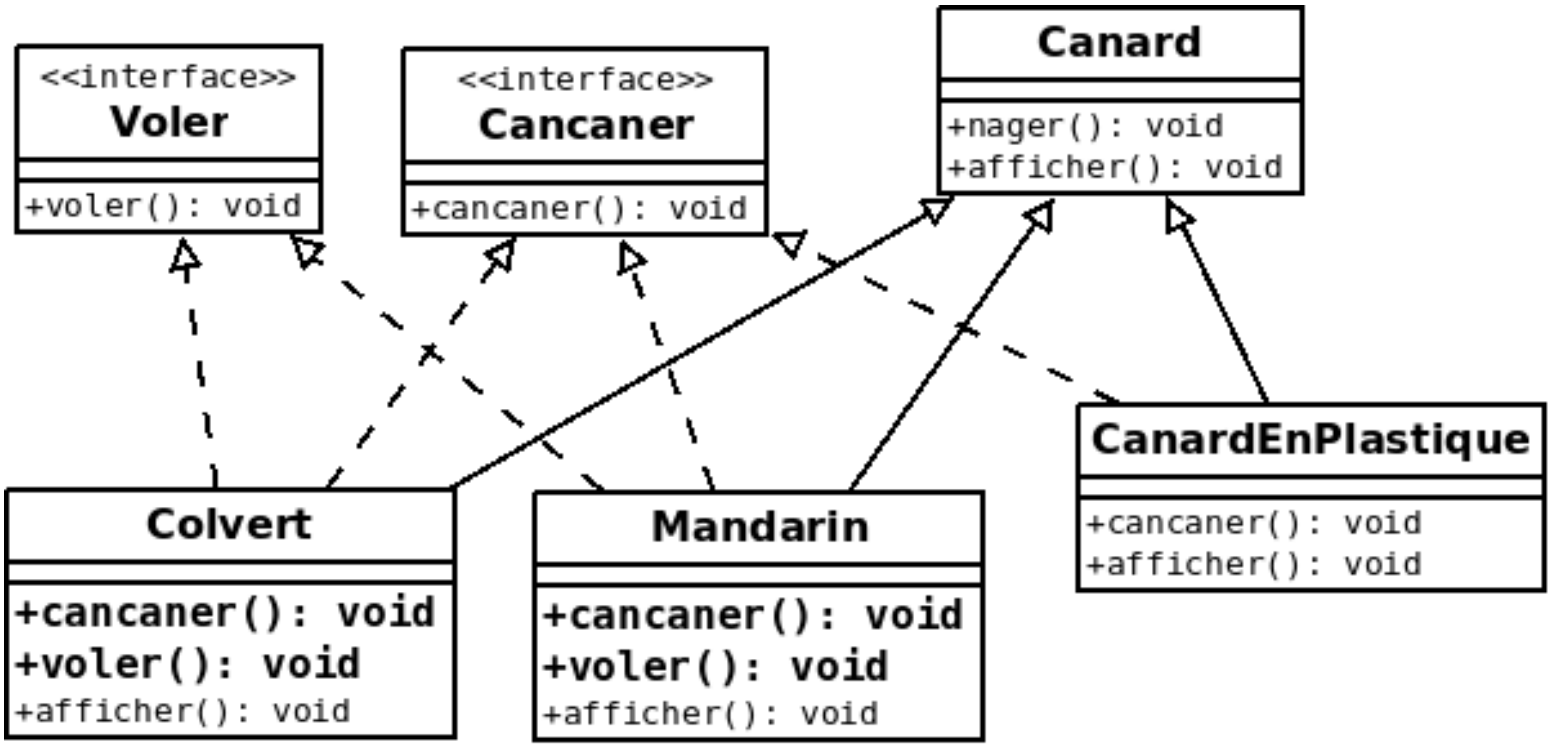
Le problème vient de la structure du programme, il faut la changer :

- Des modifications peuvent involontairement affecter des canards.
- Le code est dupliqué dans les sous-classes.
- Difficile de changer les comportements lors de l'exécution.

Nouvelle structure



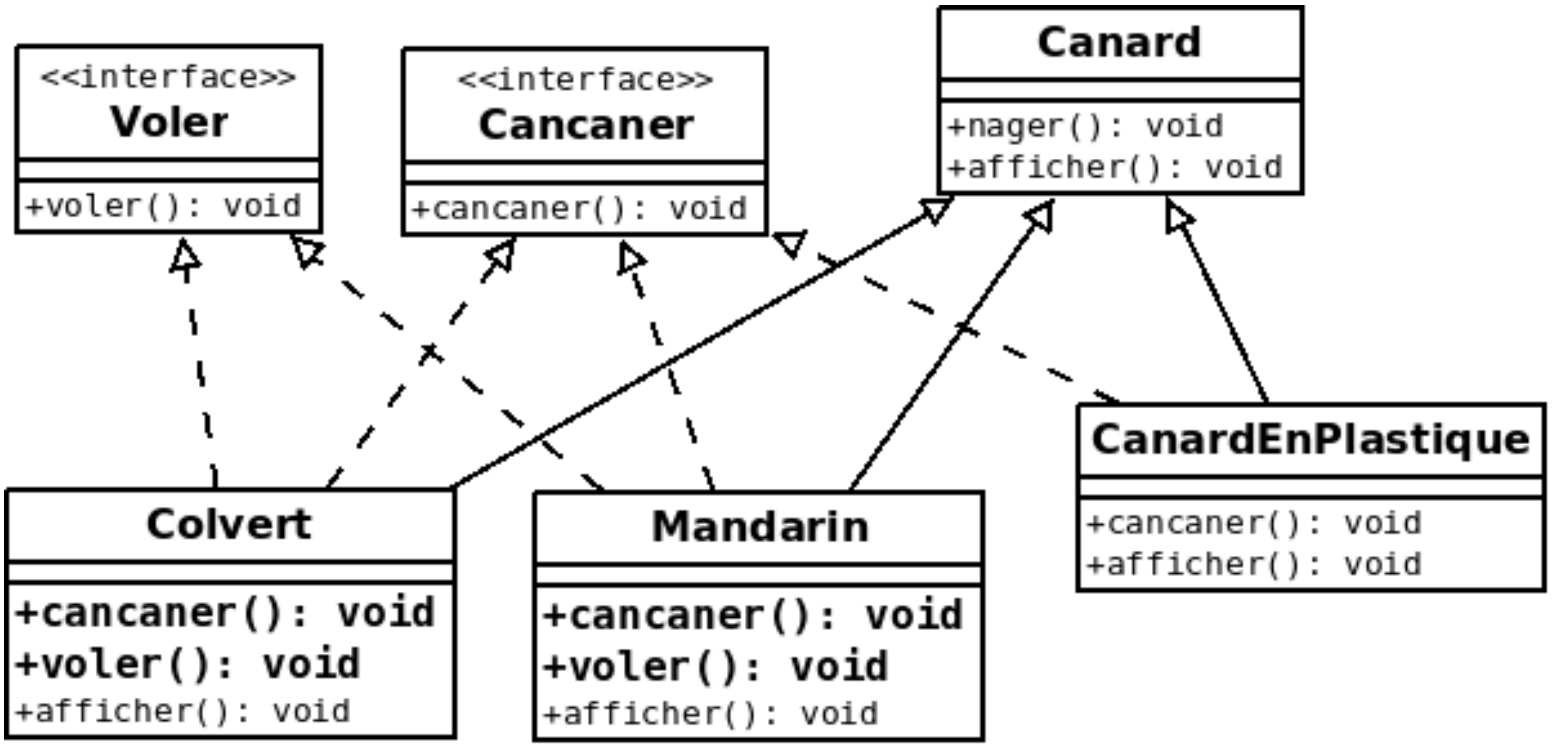
Nouvelle structure



Problème résolu ?

- Des modifications peuvent involontairement affecter des canards.
- Le code est dupliqué dans les sous-classes.
- Difficile de changer les comportements lors de l'exécution.

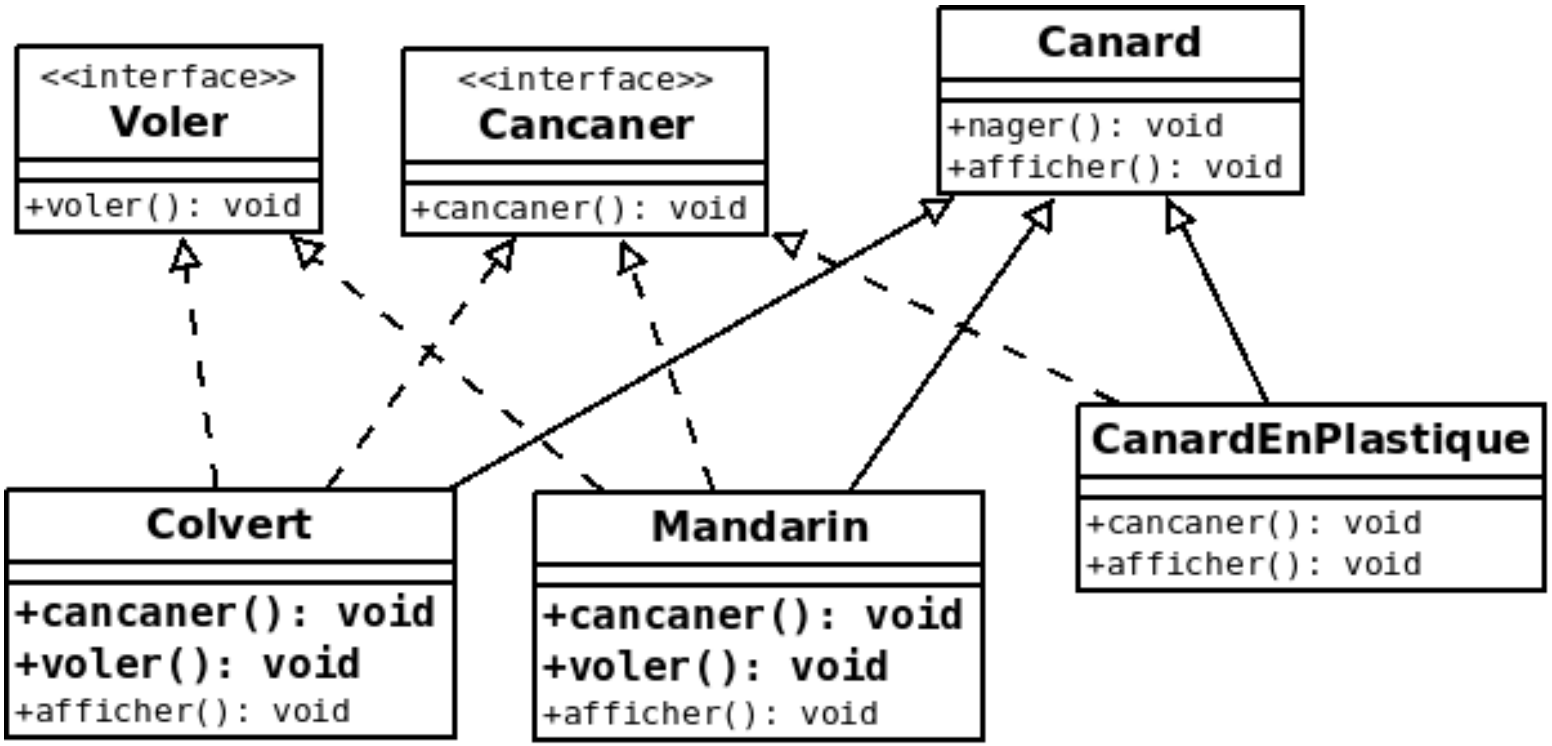
Nouvelle structure



Problème résolu ?

- ~~Des modifications peuvent involontairement affecter des canards.~~
- Le code est dupliqué dans les sous-classes.
- Difficile de changer les comportements lors de l'exécution.

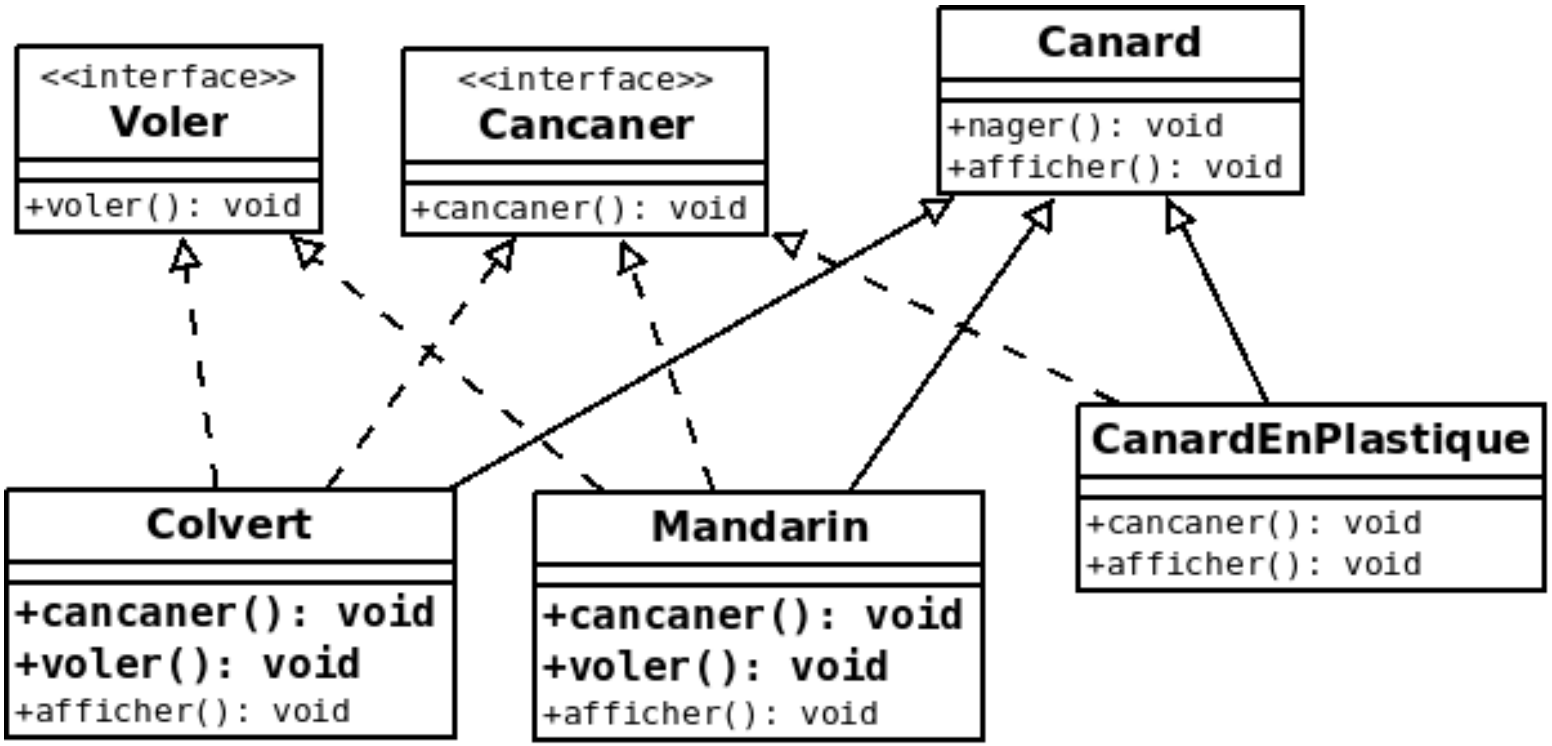
Nouvelle structure



Types

Colvert, Mandarin, etc, de type Canard ET Voler et Cancaner ?

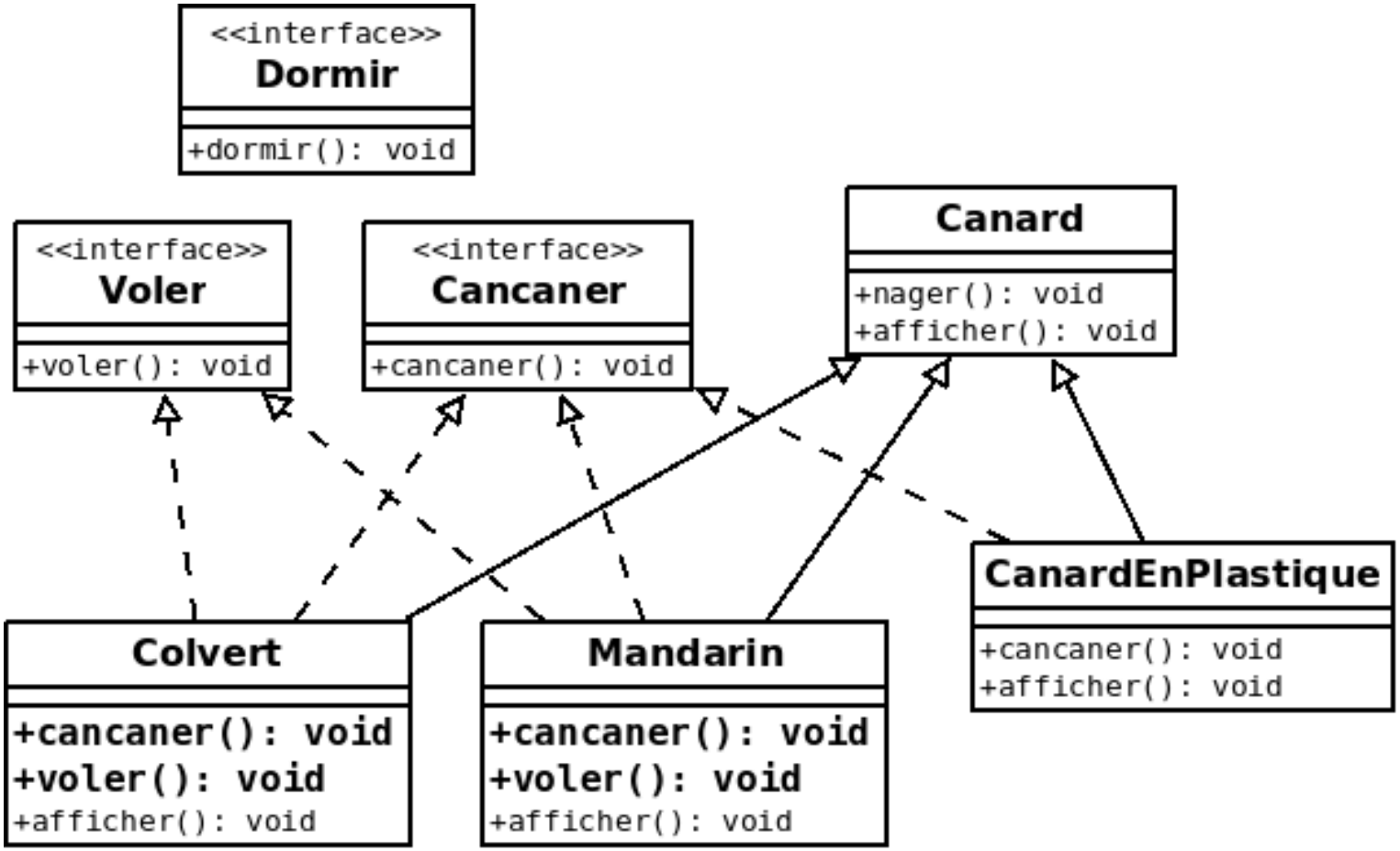
Nouvelle structure



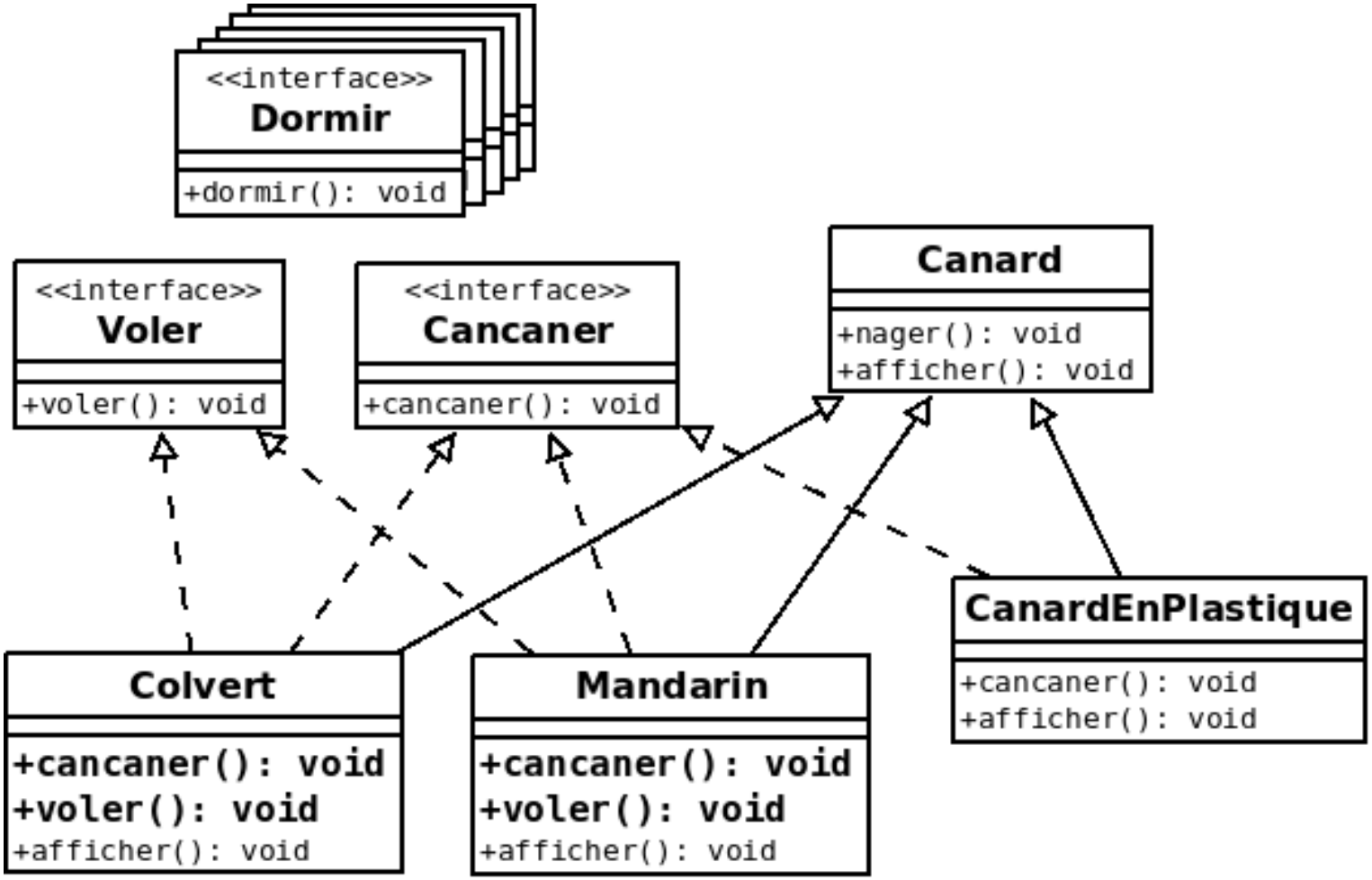
Mais surtout

On doit modifier chaque sous-classe en spécifiant le comportement d'une nouvelle fonctionnalité !

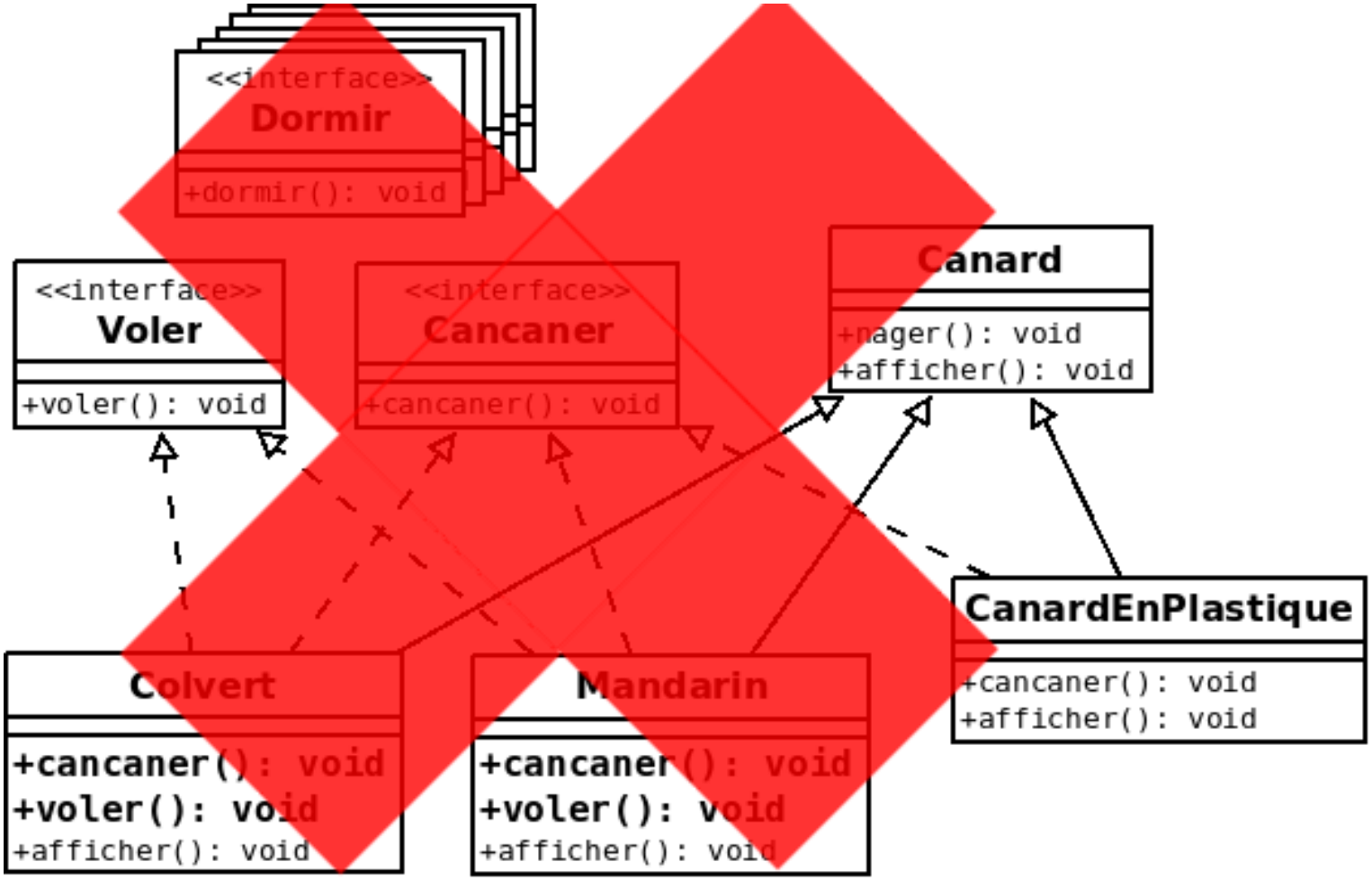
Nouvelle structure



Nouvelle structure



Nouvelle structure



La réalité des choses

Dans la vraie vie

Un programme change tout le temps :

- ajout d'une fonctionnalité,
- mise à jour,
- client qui change d'avis,
- ...

Objectif : minimiser les modifications

- Gain de temps.
- Code plus lisible.
- Moins de nouveaux bugs.

Analyser le code et le problème

Séparer ce qui peut potentiellement changer de ce qui reste constant.

Analysons. . .

But du jeu

Repérer ce qui varie et l'encapsuler (c.-à-d. le mettre dans une classe ou interface) pour l'isoler du reste du code.

$\Rightarrow$ Faire de la composition.

Ce qui varie d'un canard à l'autre

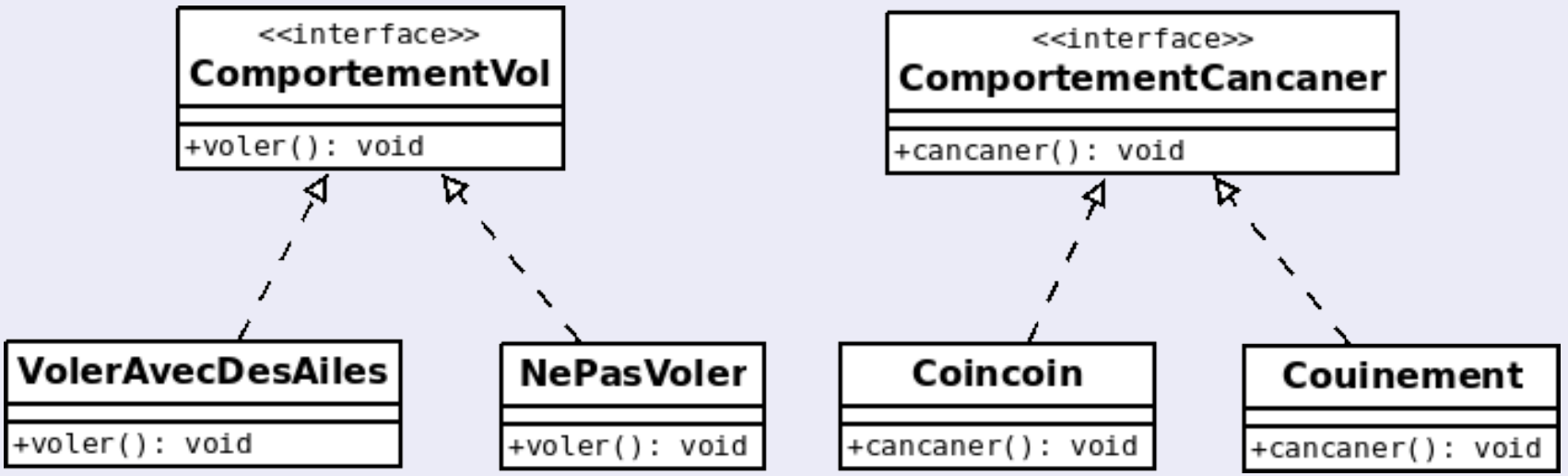
Tous les comportements: *voler()*, *cancaner()* . . .

Ce qui ne varie pas d'un canard à l'autre

afficher(), car même si ça n'affiche pas la même chose, tous les canards doivent être capable de s'afficher.

Comment encapsuler *voler()* et *cancaner()* ?

Avec de bonnes vieilles interfaces



Avec de bonnes vieilles interfaces

```
interface ComportementCancan
{
  public void cancaner();
}

class Coincoin implements ComportementCancan
{
  public void cancaner()
  {
    System.out.println("Coin Coin");
  }
}

class Couinement implements ComportementCancan
{
  public void cancaner()
  {
    System.out.println("Couine");
  }
}
```

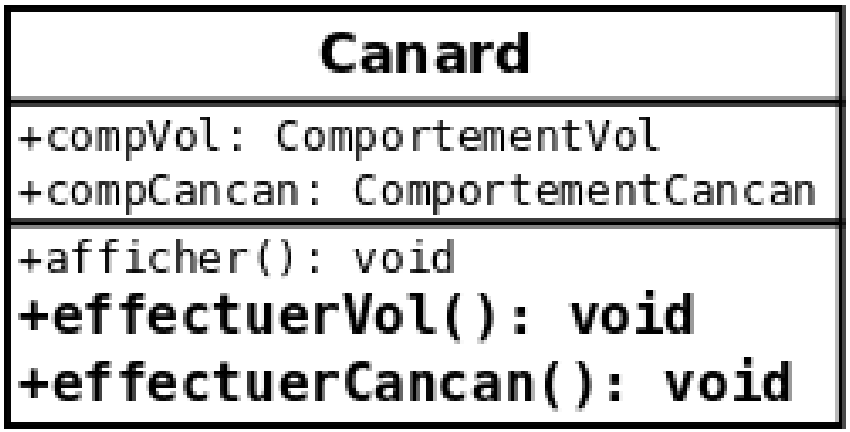
Avec de bonnes vieilles interfaces

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

class VolerAvecDesAiles implements ComportementVol
{
  public void voler()
  {
    System.out.println("Je vole.");
  }
}

class NePasVoler implements ComportementVol
{
  public void voler()
  {
    System.out.println("Je ne peux pas voler.");
  }
}
```

Intégrer ceci à notre classe Canard

Nouveau

Deux nouveaux attributs, deux nouvelles méthodes.



Intégrer ceci à notre classe Canard

En Java

```
class Canard
{
  ComportementCancan compCancan;

  public void effectuerCancan()
  {
    compCancan.cancaner(); //On délègue ce comportement
                           //à l'objet compCancan.
  }
}

class Colvert extends Canard
{
  public Colvert()
  {
    compVol = new VolerAvecDesAiles();
    compCancan = new Coincoin();
  }
}
```

Ce qui donne...

Programme

```
class Programme
{
  public static void main(String[] args)
  {
    Colvert colvert = new Colvert();
    colvert.effectuerCancan();
  }
}
```

À l'écran

```
$> java Programme
Coin Coin
```

Code facile à modifier !

Maintenant

- Facile d'ajouter une fonctionnalité sans (trop) modifier le code.
- Facile de coder le nouveau comportement d'un fonctionnalité sans changer le code existant et sans déranger les sous-classes de Canard qui ne sont pas concernées.
- Le code des comportements se trouve a un et un seul endroit !
- Facile de réutiliser les comportements pour d'autres classes.

On peut même changer des comportements à chaud !

Programme

```
class Programme
{
  public static void main(String[] args)
  {
    Colvert colvertBlesse = new Colvert();
    colvertBlesse.effectuerVol();
    // PAN !
    colvertBlesse.compVol = new NePasVoler();
    colvertBlesse.effectuerVol();
  }
}
```

À l'écran

```
$> java Programme
Je vole.
Je ne peux pas voler.
```

Moralité

Avec cette nouvelle structure

- Des modifications peuvent involontairement affecter des canards.
- Le code est dupliqué dans les sous-classes.
- Difficile de changer les comportements lors de l'exécution.

Moralité

Avec cette nouvelle structure

- ~~Des modifications peuvent involontairement affecter des canards.~~
- Le code est dupliqué dans les sous-classes.
- Difficile de changer les comportements lors de l'exécution.

Moralité

Avec cette nouvelle structure

- ~~Des modifications peuvent involontairement affecter des canards.~~
- ~~Le code est dupliqué dans les sous-classes.~~
- Difficile de changer les comportements lors de l'exécution.

Moralité

Avec cette nouvelle structure

- ~~Des modifications peuvent involontairement affecter des canards.~~
- ~~Le code est dupliqué dans les sous-classes.~~
- ~~Difficile de changer les comportements lors de l'exécution.~~

Moralité

Avec cette nouvelle structure

- ~~Des modifications peuvent involontairement affecter des canards.~~
- ~~Le code est dupliqué dans les sous-classes.~~
- ~~Difficile de changer les comportements lors de l'exécution.~~

On voit que les anciennes structures à base d'héritage n'étaient pas satisfaisantes.

Principe à garder en tête

"Favor composition over inheritance."

Vue d'ensemble

Faire le gros schéma.

Sans le savoir...

Vous venez d'apprendre votre premier pattern !

Vous venez d'apprendre votre premier pattern !

Pattern Strategy

Pattern Strategy

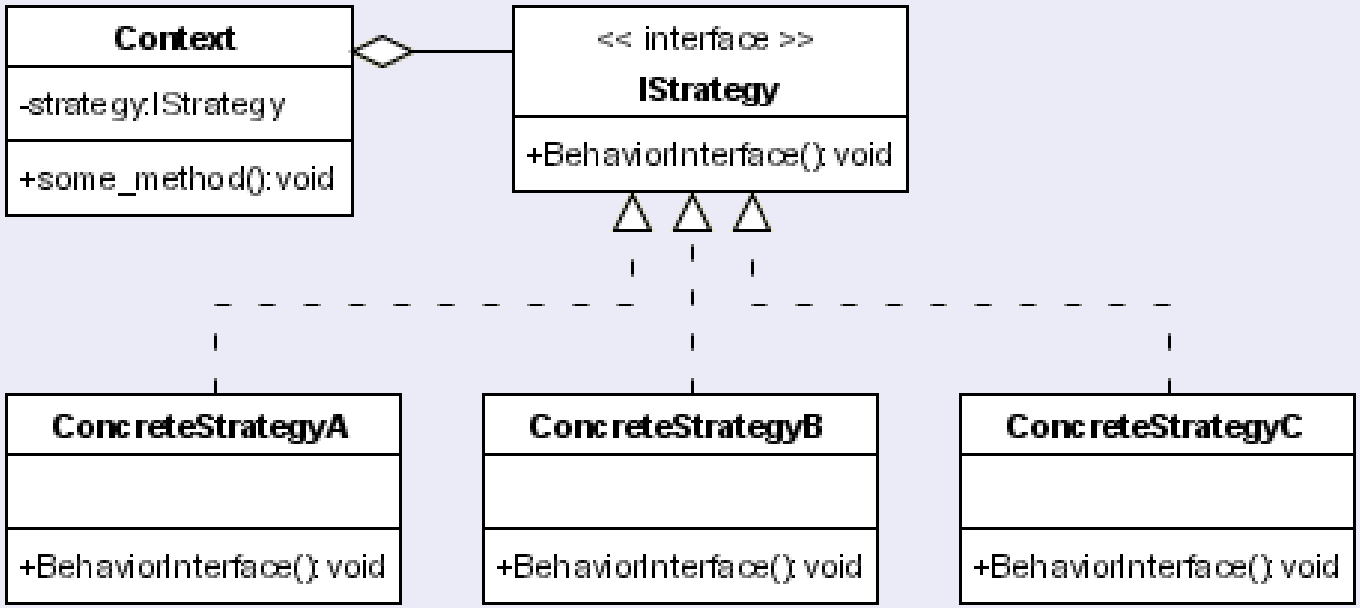
Définition

Le pattern Strategy (Stratégie) définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Modules indépendants

Dans notre exemple, le client est la classe Canard et les familles d'algorithmes sont les différentes manières de cancaner, de voler, etc.

Diagramme UML



Types de patterns

3 types de patterns

- Les patterns de **création** concernent la création de classes ou d'objets.
- Les patterns de **structure** aident à structure une composition.
- Les patterns de **comportement** définissent les interactions entre objets.

Type du pattern Strategy

Le pattern Strategy rentre dans la classe des patterns de **comportement**.

Dans ce cours

On essayera de voir 3 patterns de chaque type.

La semaine prochaine

On verra un autre pattern de comportement, un cousin éloigné de Strategy.

Le projet

Projet

Le projet

- Rapport de 10 pages max à me rendre dans mon casier le mercredi 27 novembre. **Aucun retard accepté**.
- Présentation du projet début décembre.

Le rapport

Décription rapide de votre projet. L'accent portera plutôt sur les design patterns. Décrire les patterns utilisés :

- Rapide définition,
- Pourquoi le choix de ces patterns,
- Comment s'incluent-ils dans le projet,
- Quels ont été vos difficultés,
- ...

Projet

La présentation

Son déroulement :

1. Compilation en salle machine, sous Linux.
2. Présenter la répartition des tâches. J'interroge chacun sur un des patterns mis en œuvre.
3. Dire comment interviennent les patterns dans le programme.
4. Zippage du code et envoie par email.

Important

Votre projet doit fonctionner !

Projet

Bonne idée

Commenter votre code.

Très bonne idée

Commenter votre code en utilisant doxygen.

Très mauvaise idée

Faire ni l'un ni l'autre.

Projet

1) le projet, 2) les patterns

Ne pas choisir un projet en fonction des patterns : il faut avoir la démarche inverse.

Par où commencer ?

- Coder votre projet comme si vous ne connaissiez rien des patterns.
- Une fois le projet terminé, posez-vous la question des modifications et extensions possibles. Là, vous injectez les patterns adéquats dans votre code.

Dans la vraie vie

Difficile d'attaquer un projet directement avec des patterns. En général, ils viennent après.
Commencer directement avec des patterns demande une grande expérience à la fois des patterns et du projet en lui-même !