

Un guide d'apprentissage

Tête la première Design Patterns

Évitez les erreurs de couplage gênantes



Voyez pourquoi vos amis se trompent au sujet du pattern Fabrication



Découvrez les secrets du maître des patterns



Trouvez comment le pattern Décorateur a fait grimper le prix des actions de Starbuzz Coffees



Injectez-vous directement dans le cerveau les principaux patterns



Apprenez comment la vie amoureuse de Jim s'est améliorée depuis qu'il préfère la composition à l'héritage



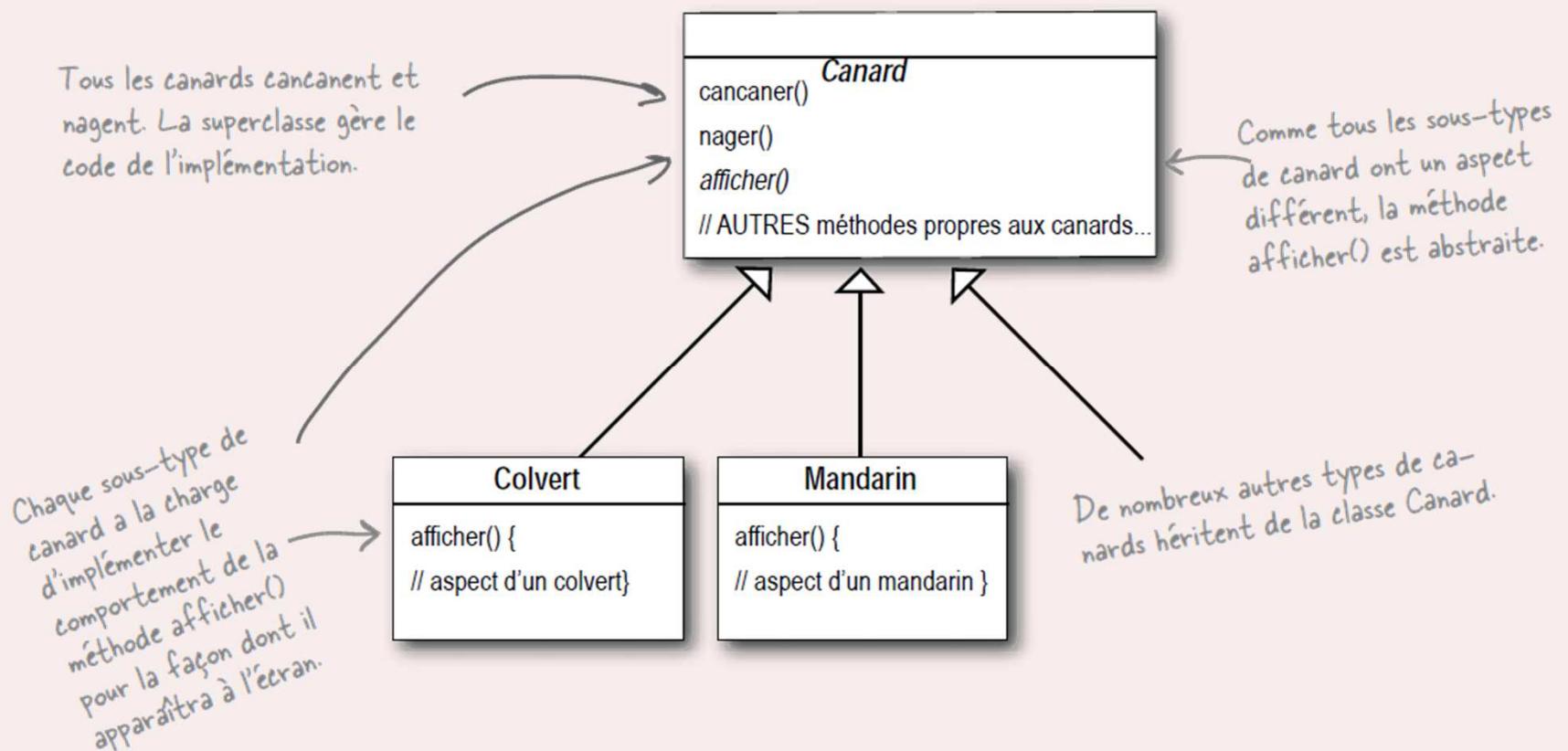
Eric Freeman & Elisabeth Freeman
avec Kathy Sierra & Bert Bates
Traduction de Marie-Cécile Baland

O'REILLY®

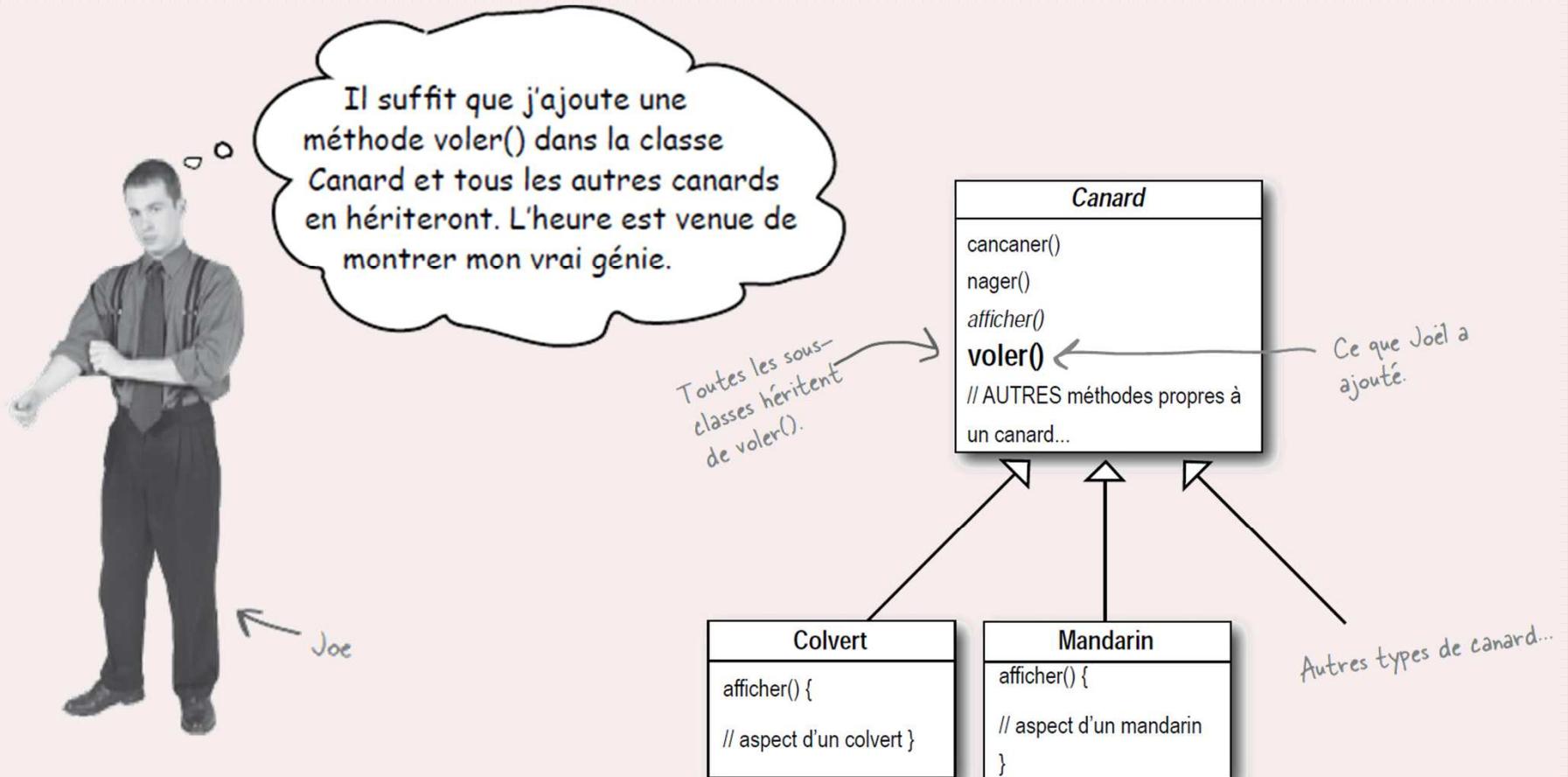
Bienvenue aux Design Patterns



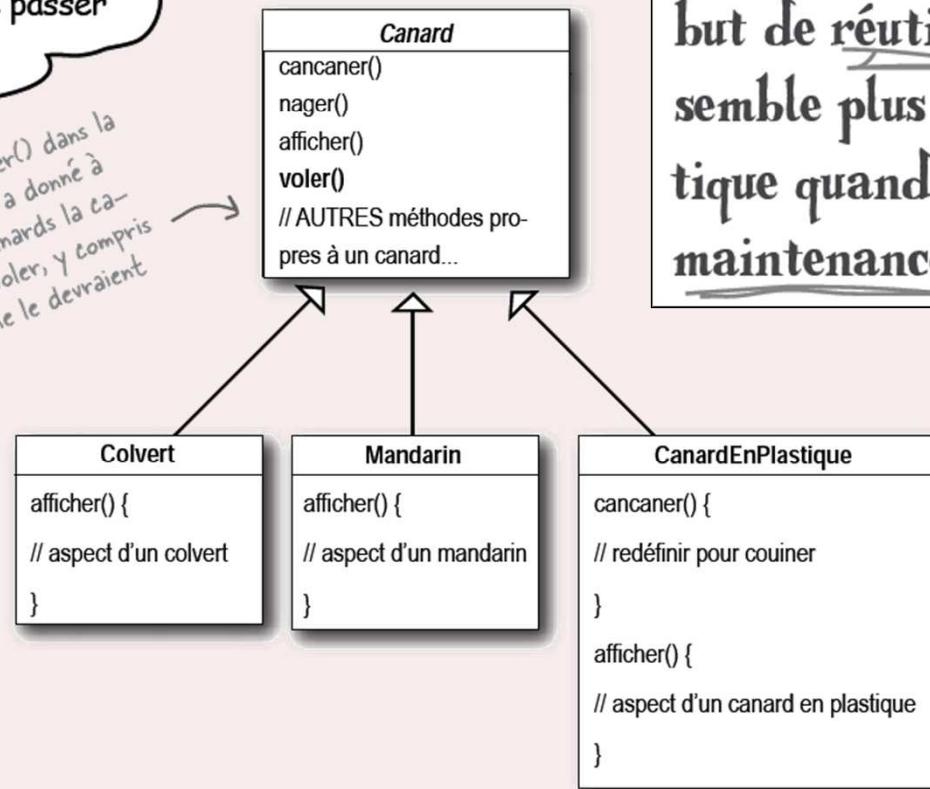
Tout a commencé par une simple application, SuperCanard



Maintenant, nous voulons que les canards volent



Il y a quelque chose qui ne va pas



Ce qu'il prenait pour une super application de l'héritage dans un but de réutilisation semble plus problématique quand il s'agit de maintenance.

Puisque les canards en plastique ne cancanent pas, cancaner() est redéfinie pour couiner.

Joe réfléchit à l'héritage...

Je pourrais toujours me contenter de redéfinir la méthode voler() dans CanardEnPlastique, comme je le fais pour cancaner()...



Mais alors qu'est-ce qui se passe quand nous ajoutons des canards leurres en bois au programme? ils ne sont pas censés voler ou cancaner ...



```
Leurre
cancaner() {
    // redéfinir pour ne rien faire
}
afficher() { // leurre}
voler() {
    // redéfinir pour ne rien faire
}
```

Voici une autre classe de la hiérarchie. Remarquez que les leurres ne volent pas plus que les canards en plastique. En outre, ils ne cancanent pas non plus.

L'héritage n'est pas la réponse

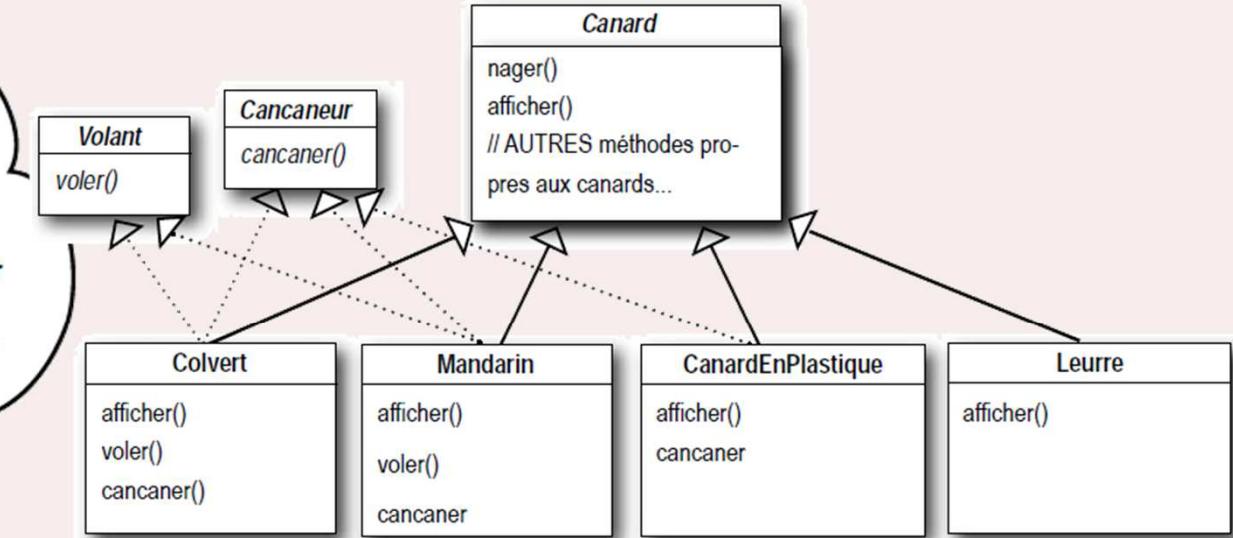
Joël vient de recevoir un mémo annonçant que les dirigeants ont décidé de réactualiser le produit tous les six mois (ils n'ont pas encore décidé comment).

Il sait que les spécifications vont changer en permanence et qu'il va peut-être être obligé de redéfinir voler() et cancaner() pour toute sous-classe de Canard qui sera ajoutée au programme... *ad vitam aeternam.*

Il a donc besoin d'un moyen plus sain pour que seuls certains types de canard (mais pas tous) puissent voler ou cancaner.

Et si nous utilisions une interface ?

Je pourrais extraire la méthode voler() de la superclasse Canard, et créer une **interface Volant()** qui aurait une méthode voler(). Ainsi, seuls les canards qui sont censés voler implémenteront cette interface et auront une méthode voler()... et je pourrais aussi créer une interface Cancanant par la même occasion, puisque tous les canards ne cancanent pas.



Et VOUS ? Que pensez-vous de cette conception ?

Et vous ? Qu'en pensez-vous ?



C'est,
comment dire... l'idée la
plus stupide que tu aies jamais eue.
Et le code dupliqué ? Si tu pensais
que redéfinir quelques méthodes était une
mauvaise idée, qu'est-ce que ça va être quand
tu devras modifier un peu le comportement
de vol dans les 48 sous-classes de
Canard qui volent ?!

À ce stade, vous attendez peut-être qu'un design pattern arrive sur son cheval blanc et vous sauve la mise. Mais où serait le plaisir ?

Non, nous allons essayer de trouver une solution à l'ancienne...
en appliquant de bons **principes de conception OO**.

Extraire ce qui varie

- Extrayez ce qui varie et « encapsulez-le » pour ne pas affecter le reste de votre code.
- Résultat ?
- Les modifications du code entraînent moins de conséquences inattendues et vos systèmes sont plus souples !



Principe de conception

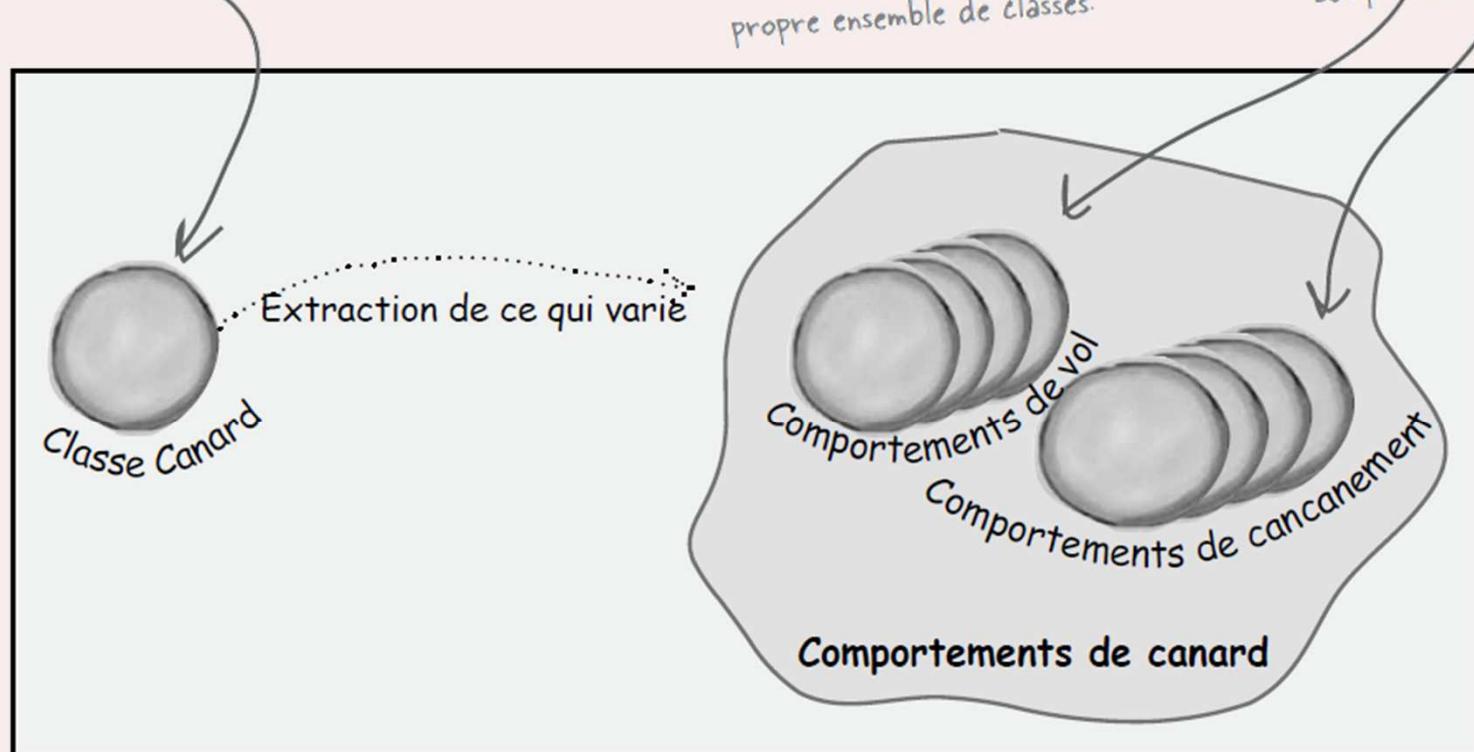
Identifiez les aspects de votre application qui varient et séparez-les de ceux qui demeurent constants

Séparer ce qui change de ce qui reste identique

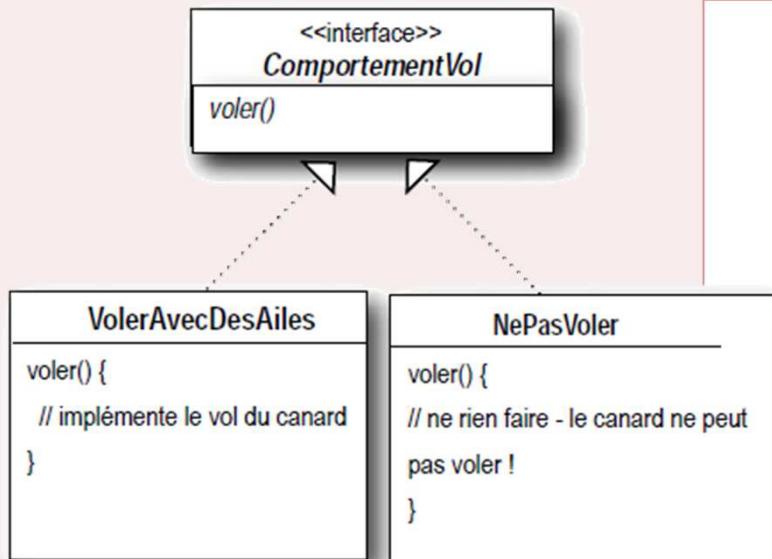
La classe Canard est toujours la superclasse de tous les canards, mais nous extrayons les comportements de vol et de cancanement et nous les plaçons dans une autre structure de classes.

Maintenant, le vol et le cancanement ont chacun leur propre ensemble de classes.

C'est là que vont résider les différentes implémentations des comportements



Conception des comportements de Canard



Les comportements de Canard résident dans une classe distincte.

- une classe qui implémente une interface comportementale particulière.

Les classes Canard ne connaissent pas l'implémentation de leurs propres comportements.



Principe de conception

Programmer une interface, non une implémentation

Programmer une interface

Programmer une implémentation :

```
Chien c = new Chien ();  
c.aboyer();
```

Programmer une interface ou un supertype :

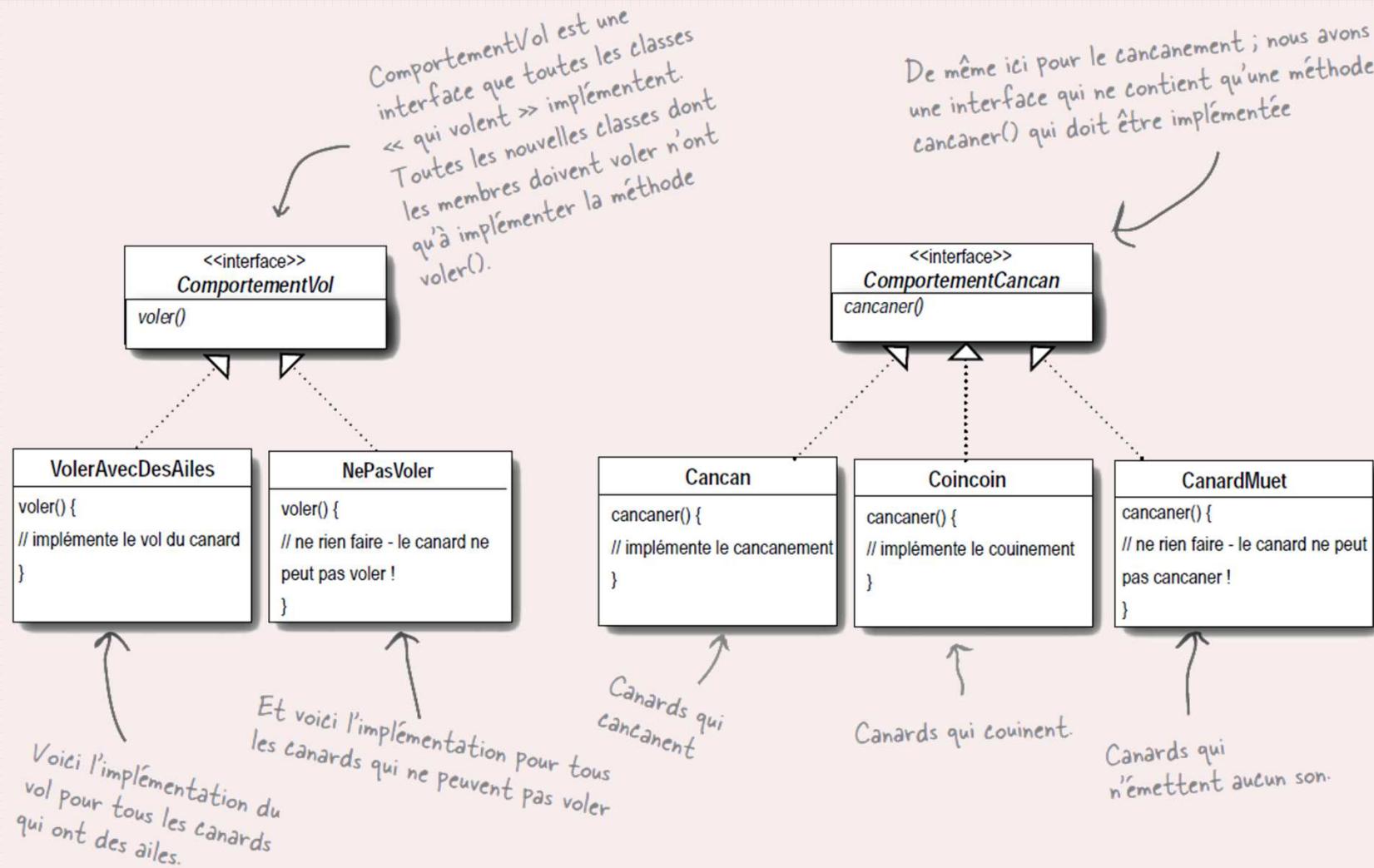
```
Animal a = new Chien ();  
a.emettreSon();
```

Mieux encore

Au lieu de coder en dur linstanciation du sous-type dans le code, affectez lobjet de limplémentation

```
Animal a = getAnimal ();  
a.emettreSon();
```

Implémenter les comportements des canards



Avantages de la conception

- Avec cette conception,
 - Les comportements de vol et de cancanement ne sont plus cachés dans nos classes Canard !
 - Les autres types d'objets peuvent réutiliser les comportements.
 - L'ajout de nouveaux comportements
 - ne modifie pas les classes comportementales existantes
 - ni aucune des classes Canard qui utilisent ses comportements.

A vos crayons



- En utilisant notre nouvelle conception, que feriez-vous pour ajouter la propulsion à réaction à l'application SuperCanard ?
Créer une classe VolAReaction qui implémente l'interface ComportementVol.
- Voyez-vous une classe qui pourrait utiliser le comportement de Cancan et qui n'est pas un canard?
Par exemple un appeau (un instrument qui imite le cri du canard).

Délégation des comportements des canards

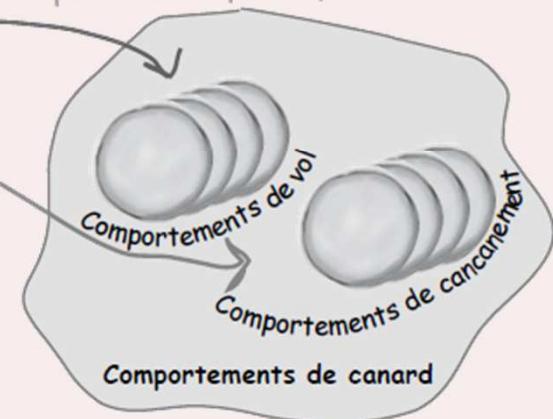
Un Canard va maintenant déléguer ses comportements.

Les variables comportementales sont déclarées du type de l'INTERFACE qui gère le comportement

Ces méthodes remplacent voler() et cancaner().

Canard
ComportementVol comportementVol
ComportementCancan comportementCancan
effectuerCancan()
nager()
afficher()
effectuerVol()
// AUTRES méthodes propres aux canards...

Lors de l'exécution, les variables d'instance contiennent une référence à un comportement spécifique.



Implémentation

- Ajouter à la classe Canard deux variables d'instance.

```
public class Canard {  
    ComportementCancan comportementCancan ;  
    ComportementVol comportementVol;  
}
```

- Implémenter maintenant effectuerCancan().

```
public void effectuerCancan() {  
    comportementCancan.cancaner();  
}
```

- Affecter les variables d'instance comportementVol et comportementCancan

```
public class Colvert extends Canard {  
    public Colvert() {  
        comportementCancan = new Cancan();  
        comportementVol = new VolerAvecDesAiles();  
    }  
}
```

Où en sommes-nous ?



Attends une seconde.
Tu n'as pas dit qu'on ne doit PAS programmer une implémentation ?
Mais qu'est-ce qu'on fait dans ce constructeur ? On crée une nouvelle instance d'une classe d'implémentation concrète de Cancan !

- Il est possible :
 - De créer de nouveaux comportements (voler, cancaner)
 - De créer de nouveaux Canard qui réutilisent des comportements déjà codés.
 - De créer des classes qui ne sont pas des Canard mais qui utilisent les comportements
- Les comportements ne sont codés qu'une seule fois
- Mais ...
 - La création des classes reste du codage d'implémentation et non d'interface.

Injection de dépendances

La façon d'affecter les variables d'instance n'est pas très satisfaisante.

Faites une pause et demandez-vous comment vous implémenteriez un canard pour que son comportement change à l'exécution.

A-UN peut être préférable à ES7-UN

- La relation A-UN est une relation intéressante :
 - Chaque canard délègue le vol ou le cancanement.
 - Lorsque vous assemblez deux classes de la sorte, vous utilisez la **composition**.
 - Au lieu d'hériter leur comportement, les canards l'obtiennent en étant composés avec le bon objet comportemental.
- La composition est utilisée dans de nombreux design patterns



Principe de conception

Péférez la composition à l'héritage

À propos des design patterns... .

Vous venez d'appliquer votre premier design pattern, le pattern STRATEGIE.

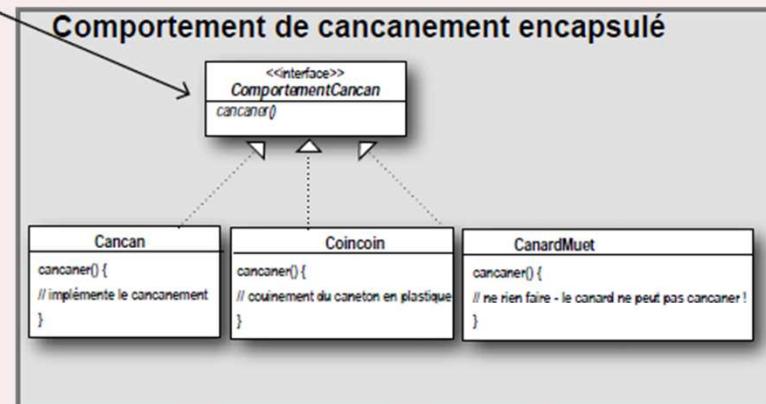
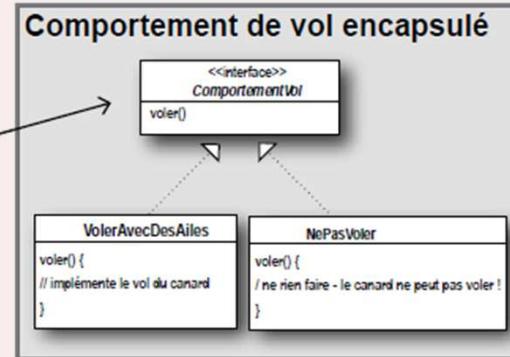
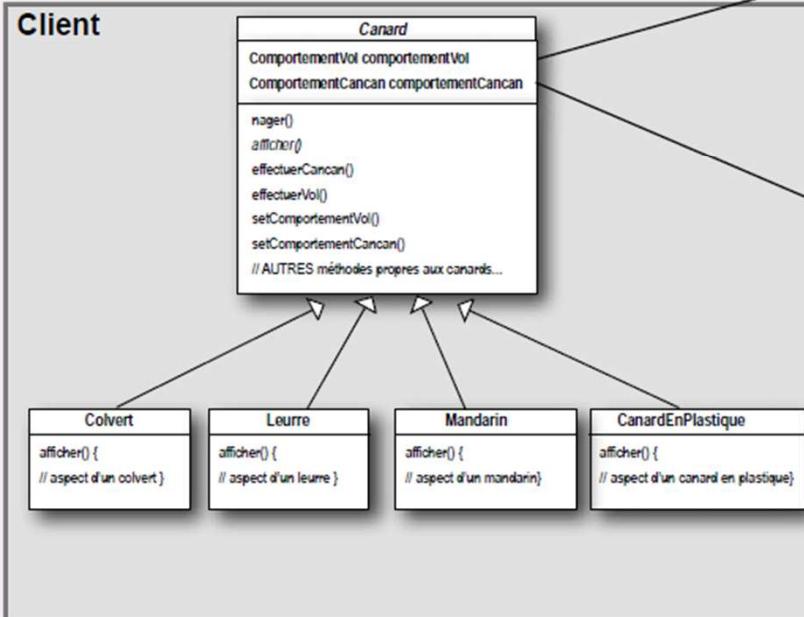


Félicitations pour votre
premier pattern !

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

Testons le code de Canard

Le client utilise une famille d'algorithmes encapsulée pour voler et cancaner.



Ces comportements sont interchangeables.