

PreReq	<ol style="list-style-type: none"> 1. Je sais programmer en Java. 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise les concepts objet de base (héritage, polymorphisme, ...).
ObjTD	Comprendre ce qu'est une conception .
Durée	1 TD et 2 TP de 1,5h (sur 2 semaines différentes).

1. L'application SuperCanard



Les exercices de ce TD sont tirés de l'excellent livre "Tête la première : Design Pattern". Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.

1.1. Application existante

Soit l'application (un jeu de simulation de mare aux canards) **SuperCanard** dont le modèle est décrit ci-dessous :

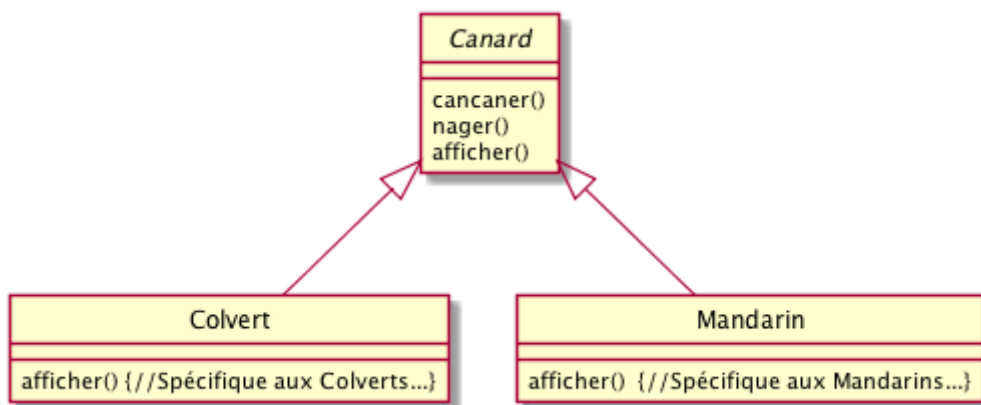


Figure 1. Extrait d'une application existante (source [plantUML](#) [ici](#))



De nombreuses autres classes héritent de **Canard**.

Voici un exemple de code :

```
abstract public class Canard {  
  
    public void cancaner() {  
        System.out.println("Je cancaner comme un Canard!");  
    }  
  
    public void nager() {  
        System.out.println("Je nage comme un Canard!");  
    }  
  
    abstract public void afficher();  
}
```

```
public class Colvert extends Canard {  
  
    public void afficher() {  
        System.out.println("Je suis un Colvert");  
    }  
  
}
```

1.2. Modification/Amélioration

Votre hiérarchie vous demande maintenant d'améliorer l'application pour être plus proche de la réalité.

Vous décidez d'ajouter une méthode `voler()` à vos canards :

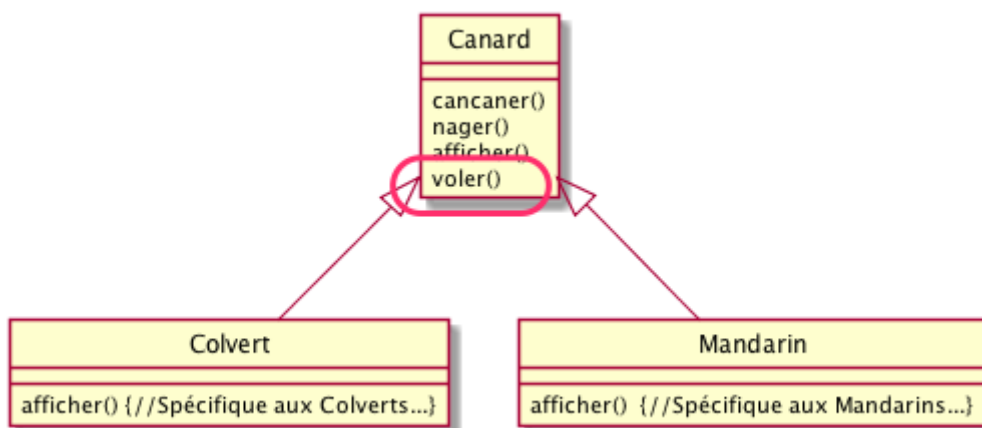


Figure 2. Nouvelle fonctionnalité

```
abstract public class Canard {  
  
    public void cancaner() {  
        System.out.println("Je cancanne comme un Canard!");  
    }  
  
    public void nager() {  
        System.out.println("Je nage comme un Canard!");  
    }  
  
    abstract public void afficher();  
  
    public void voler() {  
        System.out.println("Je vole comme un Canard!");  
    };  
}
```

1.3. Catastrophe!

La hiérarchie vous appelle en urgence : des canards en plastiques se mettent à voler dans l'application! En plus, certains canards malades, qui ne devraient pas voler, volent!



Vous avez oublié que certains canards ne volaient pas!



QUESTION

Complétez la phrase suivante : L'**héritage** c'est super pour faire de la mais c'est plus problématique pour les aspects

1.4. Solution 1 : redéfinition de méthodes

Vous songez à une première solution simple : redéfinir la méthode `voler()` dans les canards qui ne volent pas.

QUESTION

Complétez le code java suivant pour réaliser cette solution :



```
public class CanardEnPlastique extends Canard {  
  
    @Override  
    public void afficher() {  
        System.out.println("Je suis un CanardEnPlastique!");  
    }  
  
}
```

QUESTION

Dans la liste ci-après, quels sont les inconvénients à utiliser l'héritage pour définir le comportement de **Canard** ? (Plusieurs choix possibles.) :



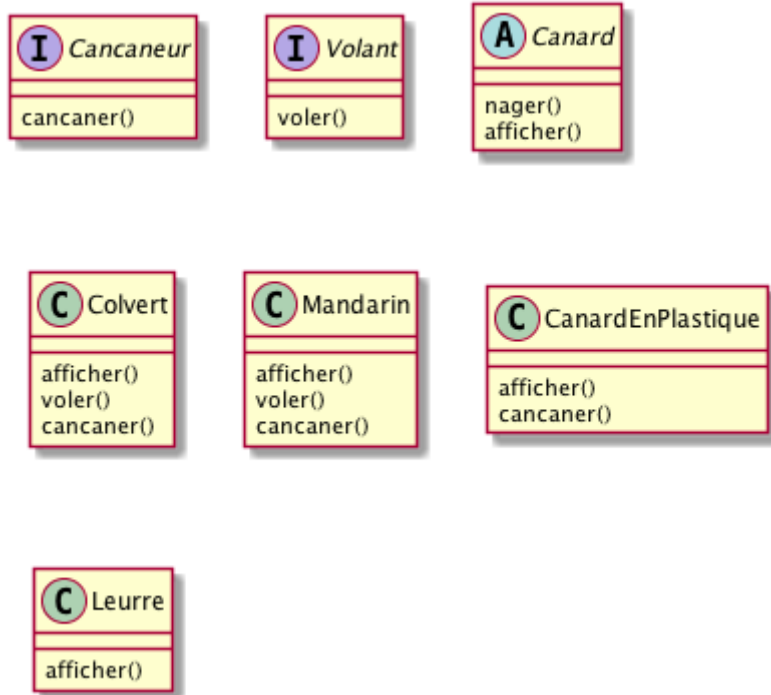
- ☐ Le même code est dupliqué (réécrit) entre les sous-classes.
- ☐ Les changements de méthodes de comportements au moment de l'exécution sont difficiles.
- ☐ Nous ne pouvons pas avoir de canards qui dansent.
- ☐ Il est difficile de connaître tous les comportements des canards
- ☐ Les canards ne peuvent pas voler et cancaner en même temps.
- ☐ Les modifications peuvent affecter involontairement d'autres canards.

1.5. Solution 2 : utilisation des interfaces

Vous songez à utiliser les interfaces pour améliorer le code.

QUESTION

1. Sur le diagramme suivant indiquez les relations d'héritage (**extends** de java) et d'implémentation (**implements** de java) :



2. Que pensez-vous de la conception obtenue ?

1.6. Solution 3 : isoler ce qui varie

Vous êtes confrontés au même problème que dans le module **MPA** de ce début d'année : **LE CHANGEMENT!**

Nous allons donc appliquer un bon Principe de conception :



Principe de conception

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



QUESTION

Quels sont les deux principales choses qui varient dans votre code?

1.6.1. Implémentation des comportements

Commençons par implémenter les comportements de manière séparée. Pour cela nous rappelons un bon principe que vous avez déjà utilisé :



Principe de conception

Programmer une interface, non une implémentation.



QUESTION

En appliquant le principe ci-dessus, proposez une conception (diagramme de classe uniquement) avec les classes et/ou interfaces (à vous de juger) suivantes : `ComportementVol`, `VolerAvecDesAiles`, `NePasVoler`.

1.6.2. Intégration du comportement

Il nous faut maintenant relier les classes de canards à leur comportement.



QUESTION

1. Ajouter à la classe `Canard` deux attributs pour référencer les comportements.
2. Enlevez les méthodes devenues inutiles.
3. Remplacez-les (donnez les implémentations) par les méthodes `effectuerVol()` et `effectuerCancan()` (qui utilisent les attributs précédents).
4. Modifiez les constructeurs de `Colvert` (par exemple) pour indiquer comment vos attributs sont initialisés.
5. Ajouter à la classe `Colvert` la méthode `setMalade()` et `setGueri()` qui permettent au run-time de modifier le comportement de vol.

1.6.3. Résumé et mise en oeuvre

Il est temps maintenant de prendre du recul et d'expérimenter les avantages de notre nouvelle conception.



QUESTION

1. Réalisez le diagramme de classe complet de l'application.
2. Que feriez-vous pour ajouter la propulsion à réaction à l'application?
3. Voyez-vous une classe qui pourrait utiliser le comportement de `Cancan` et qui n'est pas un `Canard`?

1.7. Votre premier *Design Pattern*

1.7.1. La pattern Stratégie

En fait vous venez de mettre en oeuvre votre premier *Design Pattern* : le patron *Strategy* (**Stratégie** en français).

Design pattern : Stratégie (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.

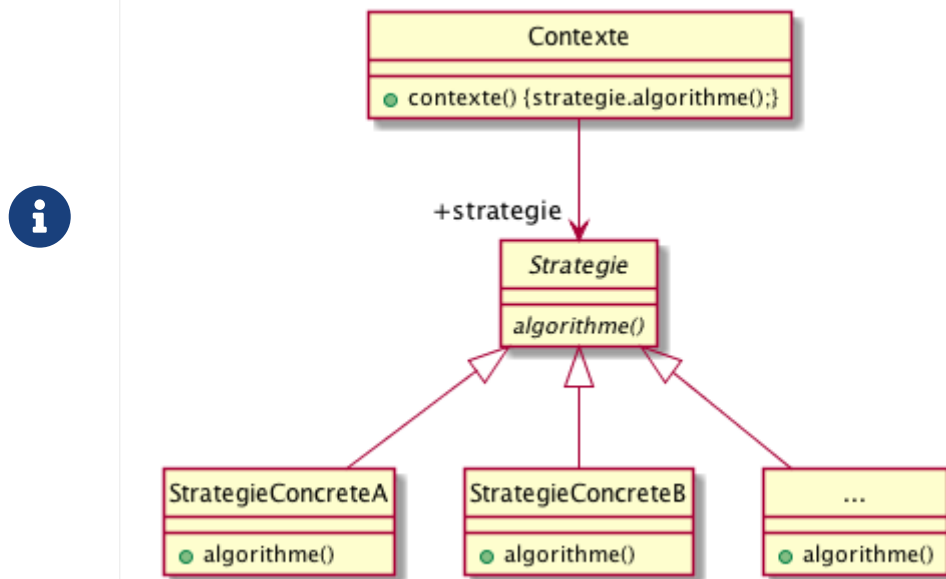


Figure 3. Modèle UML du patron Strategy

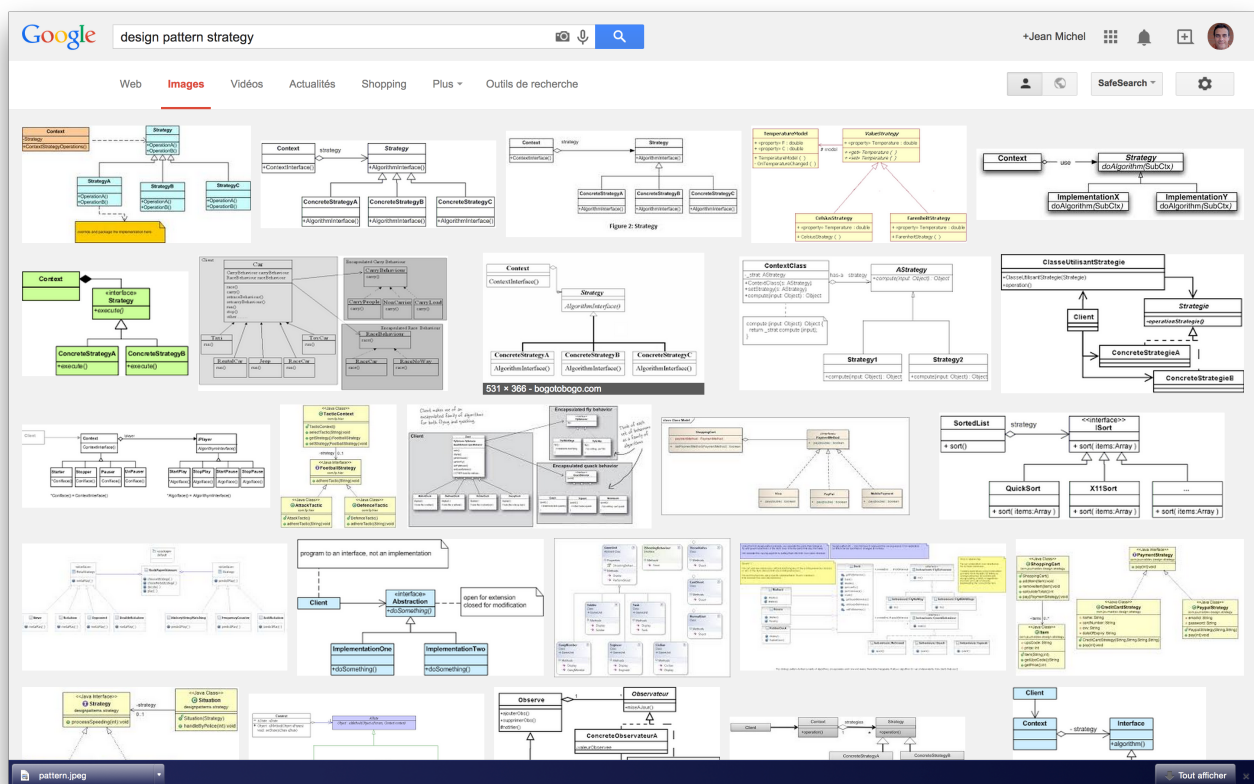


Figure 4. Quelques exemples de description du patron Strategy

1.7.2. Mise en oeuvre et révision

On vous demande de reprendre un jeu d'aventure dont seul le modèle ci-dessous est fourni.

Personnage

Reine

Roi

ComportementPoignard

ComportementArc

ComportementEpee



ComportementArme

Troll

Chevalier

1. Réorganiser les classes
2. Identifier les classes abstraites, les interfaces et les classes ordinaires.
3. Tracer les liens entre les classes ("est un", implémentation, "a un")
4. Placer la méthode `setArme()` ci-dessous :

```
setArme(ComportementArme a) {  
    this.arme = a;  
}
```

Pour aller plus loin

Nous avons utilisé sans le nommer un troisième bon principe :



Principe de conception

Préférez la composition à l'héritage.

QUESTION

Quelle différence entre notre conception finale et une implémentation du type :



```
abstract public class Canard implements ComportementVol {...}
```



QUESTION

Comment testeriez-vous la mise en oeuvre du patron [Strategy](#)?



N'hésitez pas à consulter un autre exemple, orienté "jeux de rôle", [ici \(p. 116\)](#).