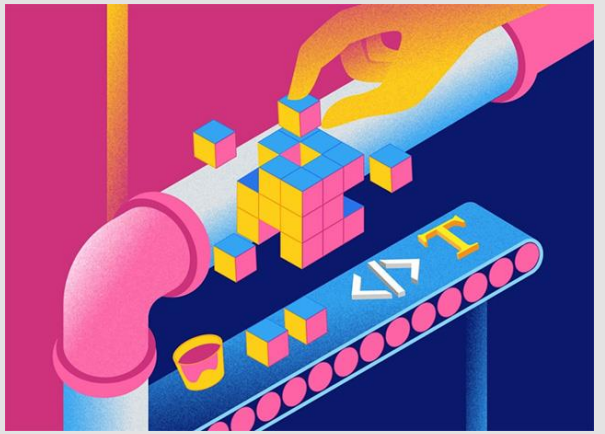
ENSET
ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA
المدرسة العليا لأساتذة التعليم التقني المحمدية

# Design Patterns

**2 ème année Cycle Ingénieur**

Pr. SARA RETAL

**GLSID 2, ICCN 2 & IIBDCC 2**




# Objectifs

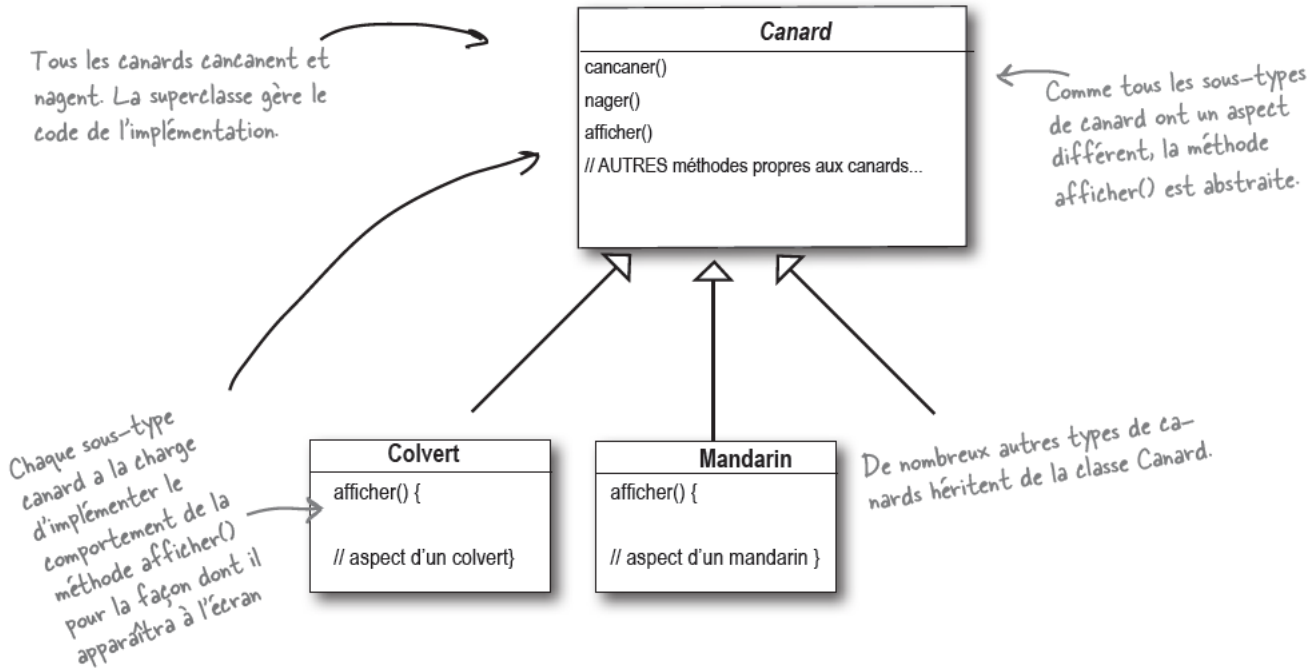
Exploiter l'expérience et les leçons tirées par d'autres développeurs qui ont déjà suivi le même chemin, rencontré les mêmes problèmes de conception et survécu au voyage.

Voir l'usage et les avantages des design patterns, revoir quelques principes fondamentaux de la conception OO et étudier un exemple de fonctionnement d'un pattern.

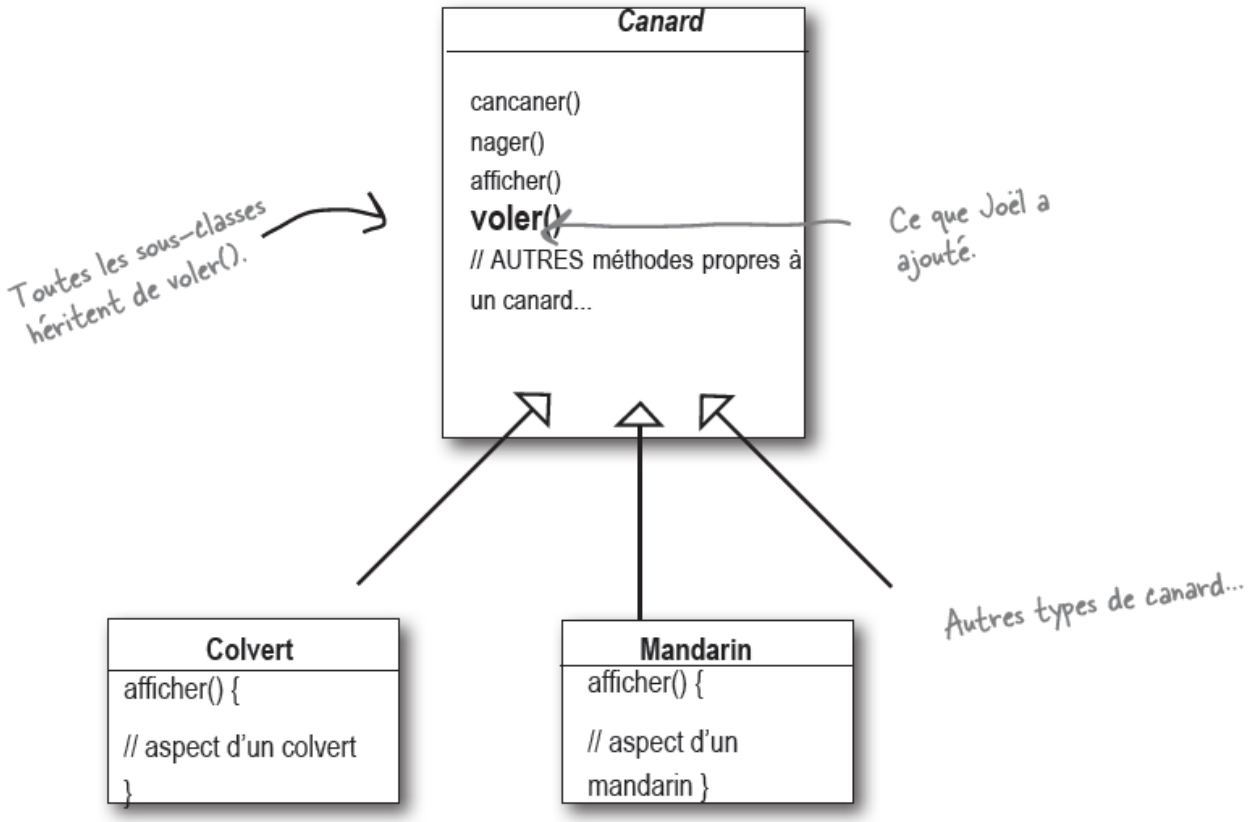
La meilleure façon d'utiliser un pattern est de le *charger dans votre cerveau* puis de *reconnaître* les points de vos conceptions et des applications existantes auxquels vous pouvez *les appliquer*. Au lieu de réutiliser du *code*, les patterns vous permettent de réutiliser de l'*expérience*.

# Exemple

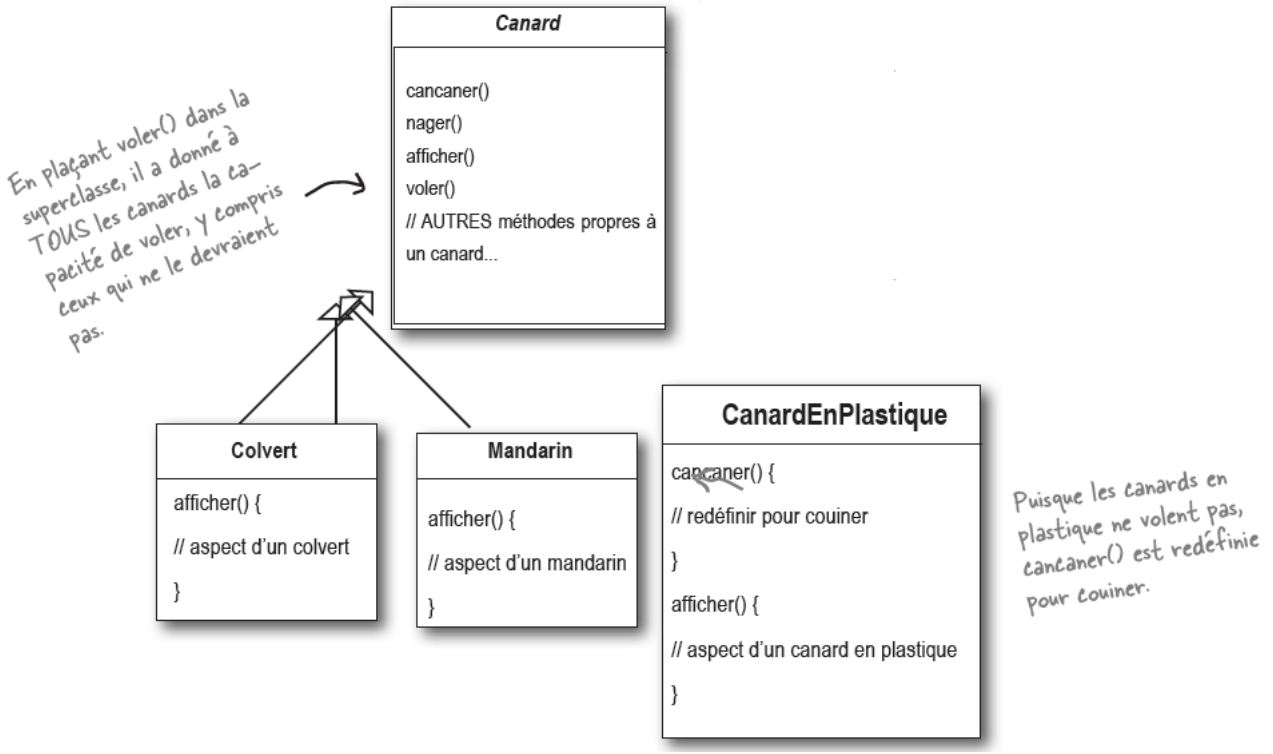
Joël travaille pour une société qui a rencontré un énorme succès avec un jeu de simulation de mare aux canards, *SuperCanard*. Le jeu affiche toutes sortes de canards qui nagent et émettent des sons. Les premiers concepteurs du système ont utilisé des techniques OO standard et créé une superclasse Canard dont tous les autres types de canards héritent.



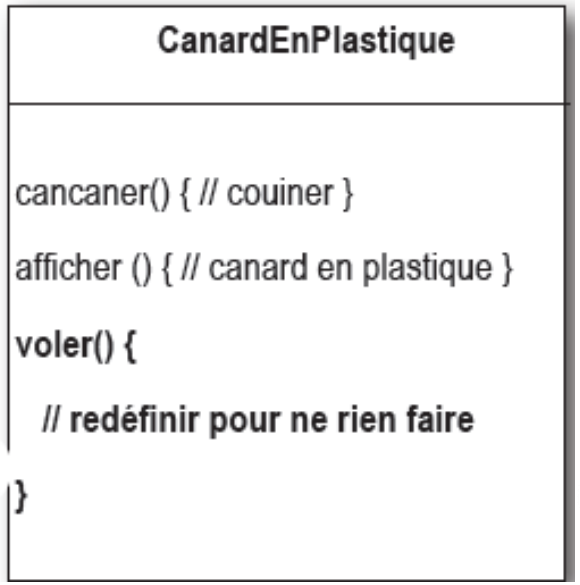
# Exemple


Canard
cancaner()
nager()
afficher()
voler()
// AUTRES méthodes propres à un canard...
Toutes les sous-classes héritent de voler().
Ce que Joël a ajouté.
Colvert
afficher() {
// aspect d'un colvert
}
Mandarin
afficher() {
// aspect d'un mandarin }
Autres types de canard...

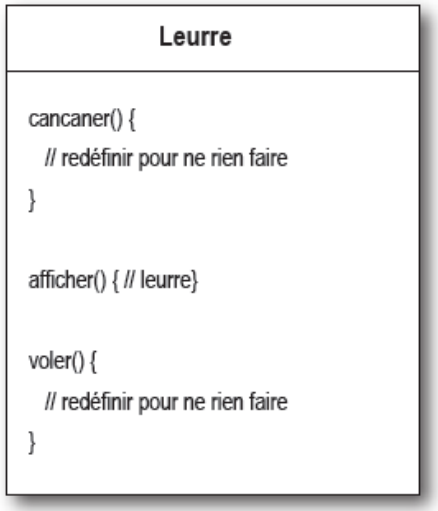
# Exemple


En plaçant voler() dans la superclasse, il a donné à TOUS les canards la capacité de voler, y compris ceux qui ne le devraient pas.
Canard
cancaner()
nager()
afficher()
voler()
// AUTRES méthodes propres à un canard...
Colvert
afficher() {
// aspect d'un colvert
}
Mandarin
afficher() {
// aspect d'un mandarin
}
CanardEnPlastique
cancaner() {
// redéfinir pour couiner
}
afficher() {
// aspect d'un canard en plastique
}
Puisque les canards en plastique ne volent pas, cancaner() est redéfinie pour couiner.

# Exemple




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.



# Exemple

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

❑ A. Le code est dupliqué entre les sousclasses.

❑ B. Les changements de comportement au

moment de l'exécution sont difficiles

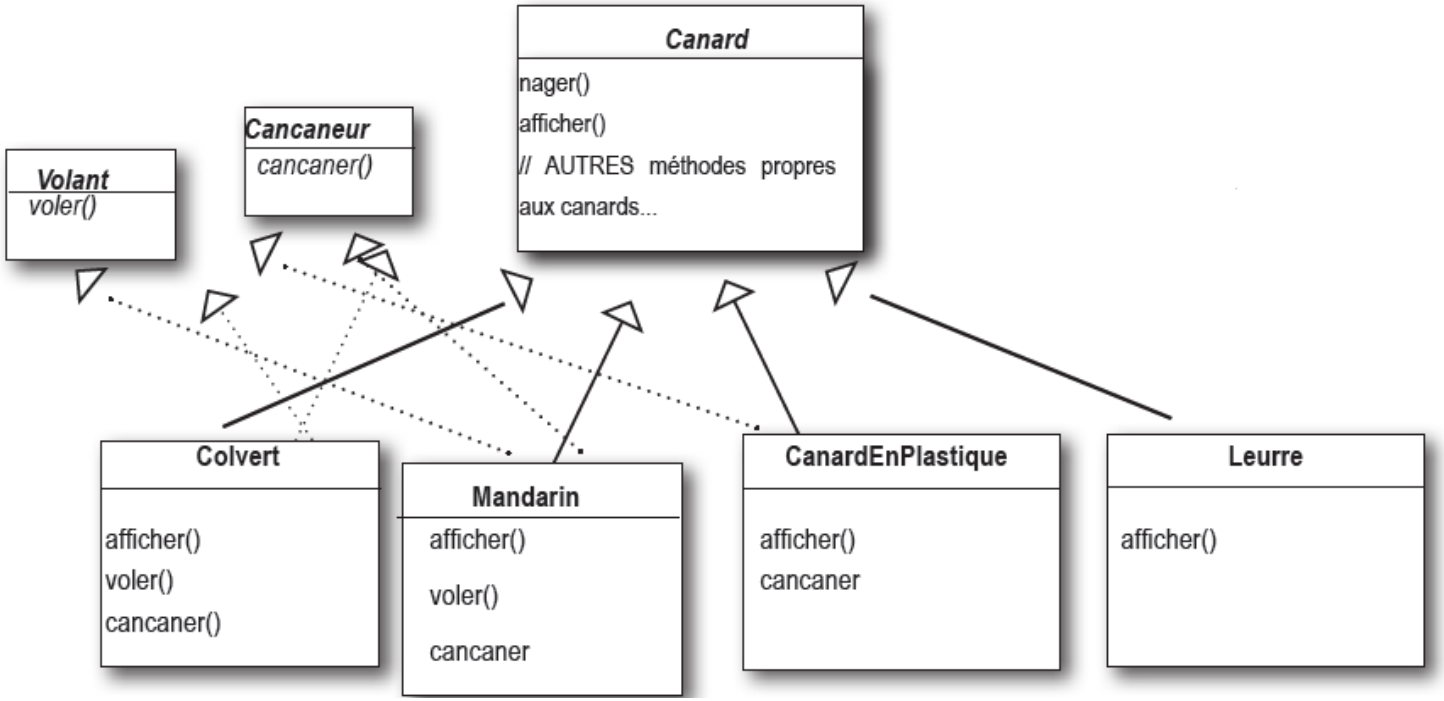
❑ C. Nous ne pouvons pas avoir de canards qui dansent.

❑ D. Il est difficile de connaître tous les comportements des canards.

❑ E. Les canards ne peuvent pas voler et cancaner en même temps.

❑ F. Les modifications peuvent affecter involontairement d'autres canards.

# Exemple


Canard
nager()
afficher()
// AUTRES méthodes propres aux canards...
Cancaneur
cancaner()
Volant
voler()
Colvert
afficher()
voler()
cancaner()
Mandarin
afficher()
voler()
cancaner
CanardEnPlastique
afficher()
cancaner
Leurre
afficher()

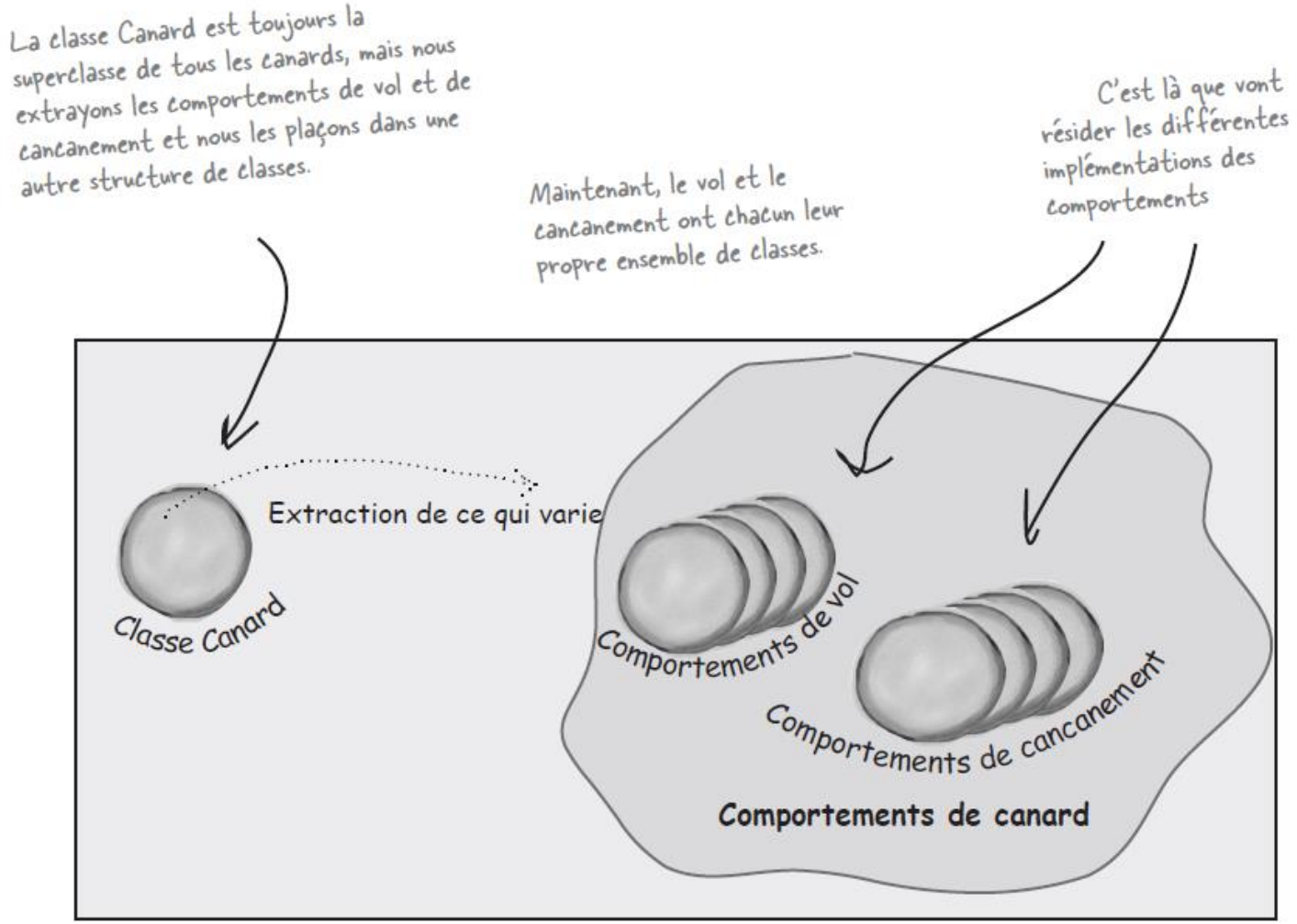
# Principe de conception

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

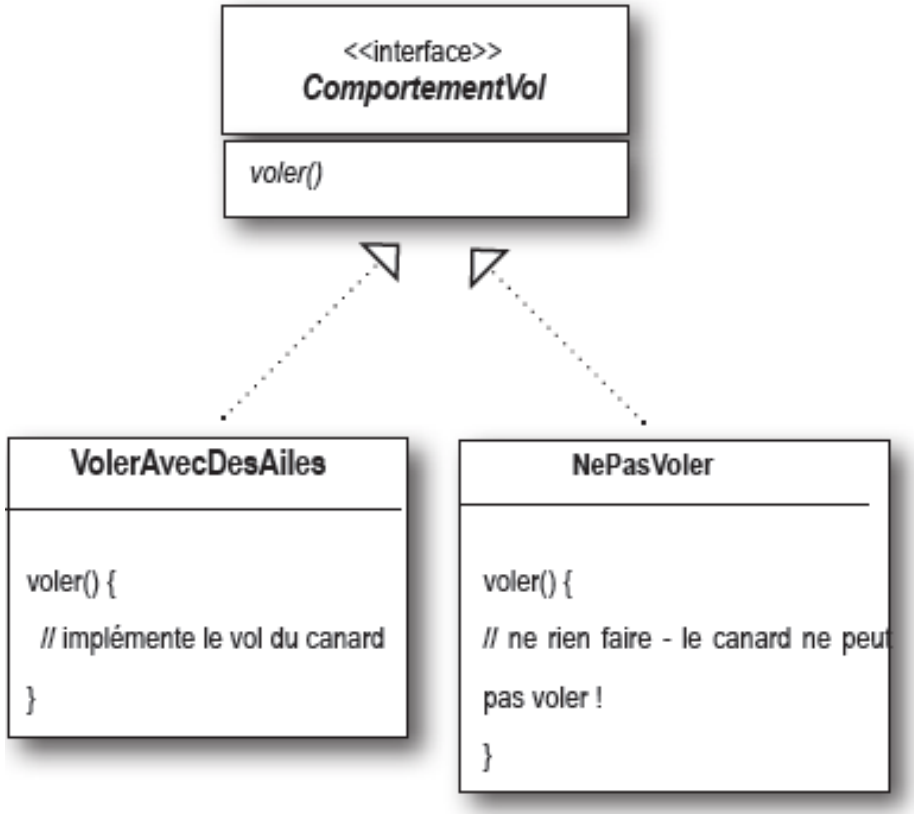
Extrayez ce qui varie et « encapsulez-le » pour ne pas affecter le reste de votre code. Résultat ?

Les modifications du code entraînent moins de conséquences inattendues et vos systèmes sont plus souples !

# Exemple


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
Extraction de ce qui varie
Classe Canard
Comportements de vol
Comportements de cancanement
Comportements de canard

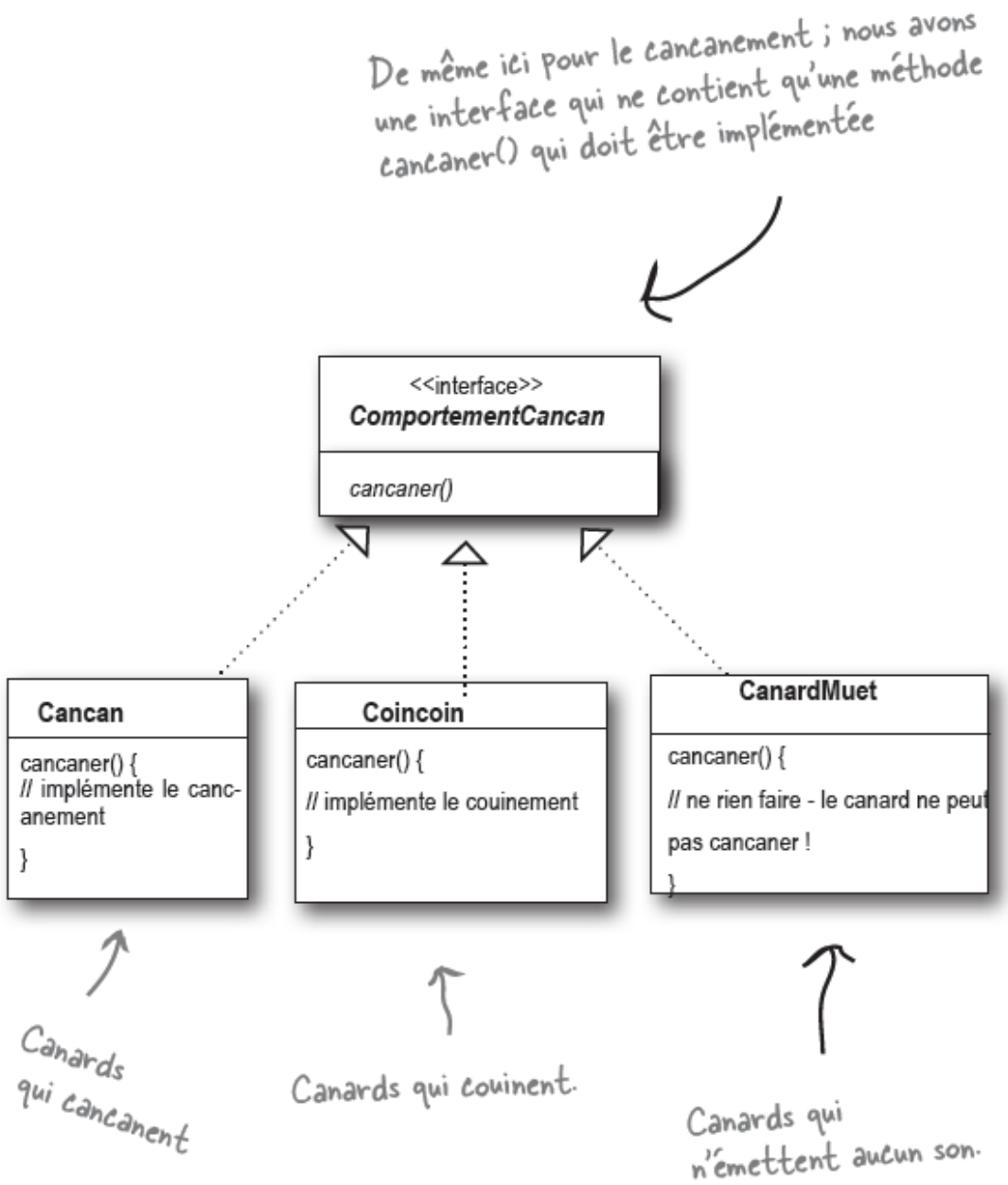
# Exemple


<<interface>>
ComportementVol
voler()
VolerAvecDesAiles
voler() {
// implémente le vol du canard
}
NePasVoler
voler() {
// ne rien faire - le canard ne peut
pas voler !
}

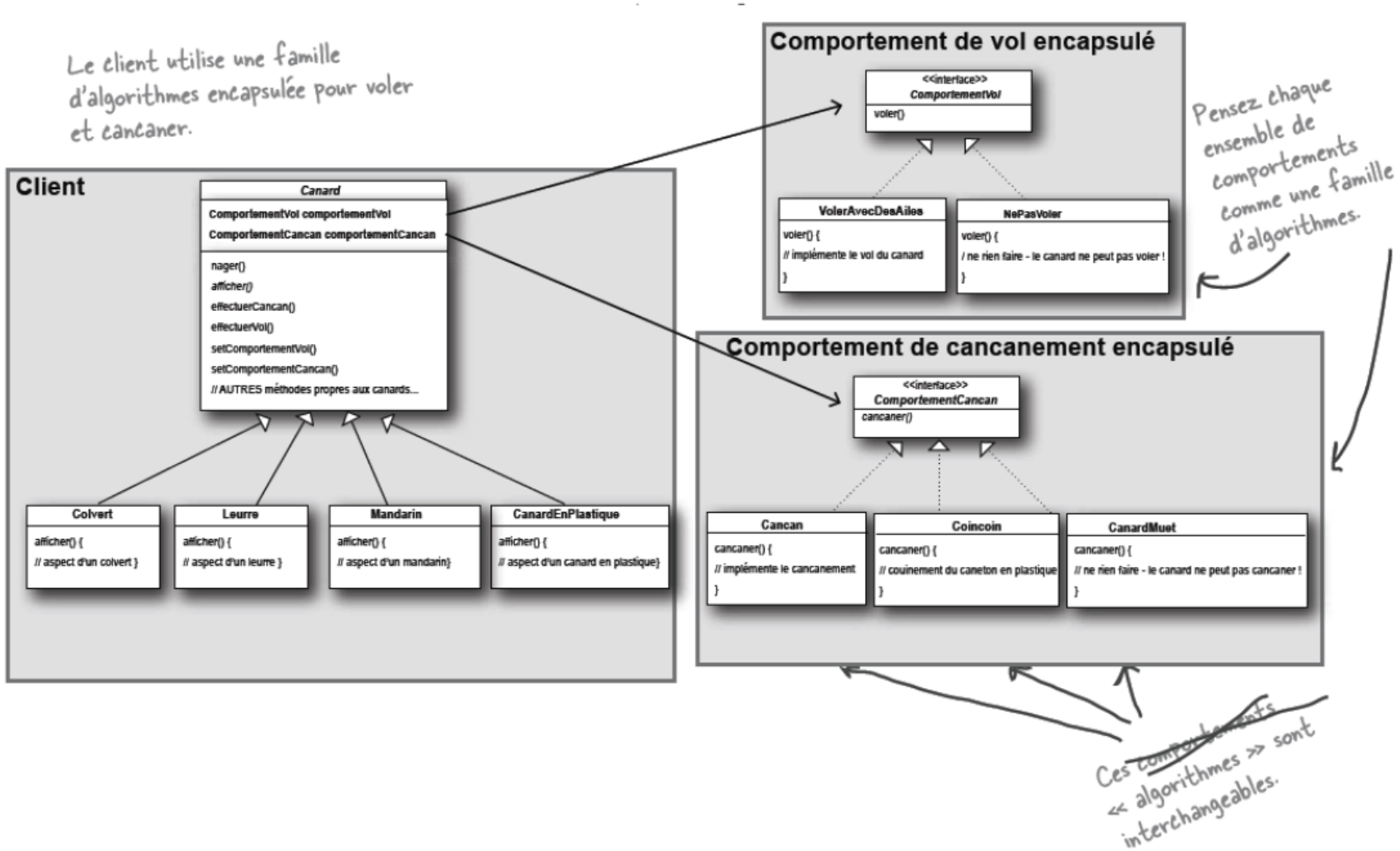
# Exemple

Le type déclaré des variables doit être un supertype, généralement une interface ou une classe abstraite. Ainsi, les objets affectés à ces variables  peuvent être n'importe quelle implémentation concrète du supertype, ce qui signifie que la classe qui les déclare n'a pas besoin de savoir quels sont les types des objets réels ! »

# Exemple


De même ici pour le cancanement ; nous avons une interface qui ne contient qu'une méthode cancaner() qui doit être implémentée
<<interface>>
ComportementCancan
cancaner()
Cancan
cancaner() {
// implémente le canc-anement
}
Coincoin
cancaner() {
// implémente le couinement
}
CanardMuet
cancaner() {
// ne rien faire - le canard ne peut pas cancaner !
}
Canards qui cancanent
Canards qui couinent.
Canards qui n'émettent aucun son.

# Exemple


Le client utilise une famille d'algorithmes encapsulée pour voler et cancaner.
Client
Canard
ComportementVol comportementVol
ComportementCancan comportementCancan
nager()
afficher()
effectuerCancan()
effectuerVol()
setComportementVol()
setComportementCancan()
// AUTRES méthodes propres aux canards...
Colvert
afficher() {
// aspect d'un colvert }
Leurre
afficher() {
// aspect d'un leurre }
Mandarin
afficher() {
// aspect d'un mandarin}
CanardEnPlastique
afficher() {
// aspect d'un canard en plastique}
Comportement de vol encapsulé
<<interface>>
ComportementVol
voler()
VolerAvecDesAiles
voler() {
// implémente le vol du canard
}
NePasVoler
voler() {
// ne rien faire - le canard ne peut pas voler !
}
Pensez chaque ensemble de comportements comme une famille d'algorithmes.
Comportement de cancanement encapsulé
<<interface>>
ComportementCancan
cancaner()
Cancan
cancaner() {
// implémente le cancanement
}
Coincoin
cancaner() {
// couinement du caneton en plastique
}
CanardMuet
cancaner() {
// ne rien faire - le canard ne peut pas cancaner !
}
Ces comportements << algorithmes >> sont interchangeables.

# Exemple

```
public abstract class Canard {

    ComportementVol comportementVol;
    ComportementCancan comportementCancan;
    public Canard() {
    }

    public abstract void afficher();

    public void effectuerVol() {
      comportementVol.voler();
    }

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

    public void nager() {
    System.out.println("Tous les canards flottent, même les leurres!");
    }
}
```

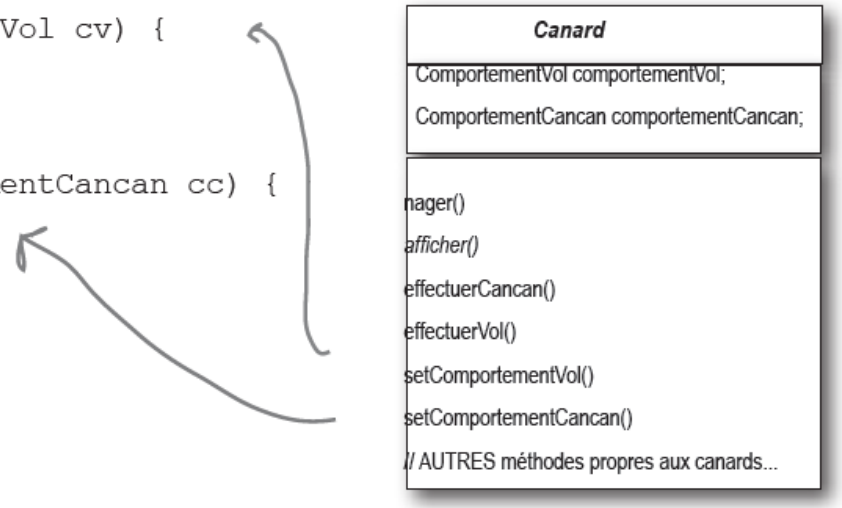
Déclare deux variables de référence pour les types des interfaces comportementales. Toutes les sous-classes de Canard (dans le même package) en héritent.

Délègue à la classe comportementale.

# Exemple

```
public void setComportementVol(ComportementVol cv) {
    comportementVol = cv;
}

public void setComportementCancan(ComportementCancan cc) {
    comportementCancan = cc;
}
```




Nous pouvons appeler ces méthodes chaque fois que nous voulons modifier le comportement d'un canard à la volée.

# Exemple

```
public class PrototypeCanard extends Canard {
    public PrototypeCanard() {
        comportementVol = new NePasVoler();
        comportementCancan = new Cancan();
    }

public void afficher() {
        System.out.println("Je suis un prototype de canard");
    }
}
```

Notre nouveau canard vient au monde...sans aucun moyen de voler.

# Exemple

Qu'à cela ne tienne ! Nous créons un nouveau comportement de vol : la propulsion à réaction

```
public class PropulsionAReaction implements ComportementVol {
    public void voler() {
        System.out.println("Je vole avec un réacteur !");
    }
}
```

# Exemple

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

            Canard proto = new PrototypeCanard();
            proto.effectuerVol();
            proto.setComportementVol(new PropulsionAReaction());
            proto.effectuerVol();
    }
}
```

Le premier appel de effectuerVol() délègue à l'objet comportementVol défini dans le constructeur de PrototypeCanard, qui est une instance de NePasVoler.

Ceci invoque la méthode set héritée du prototype, et ...voilà ! Le prototype est soudain doté d'une fonctionnalité de vol... à réaction

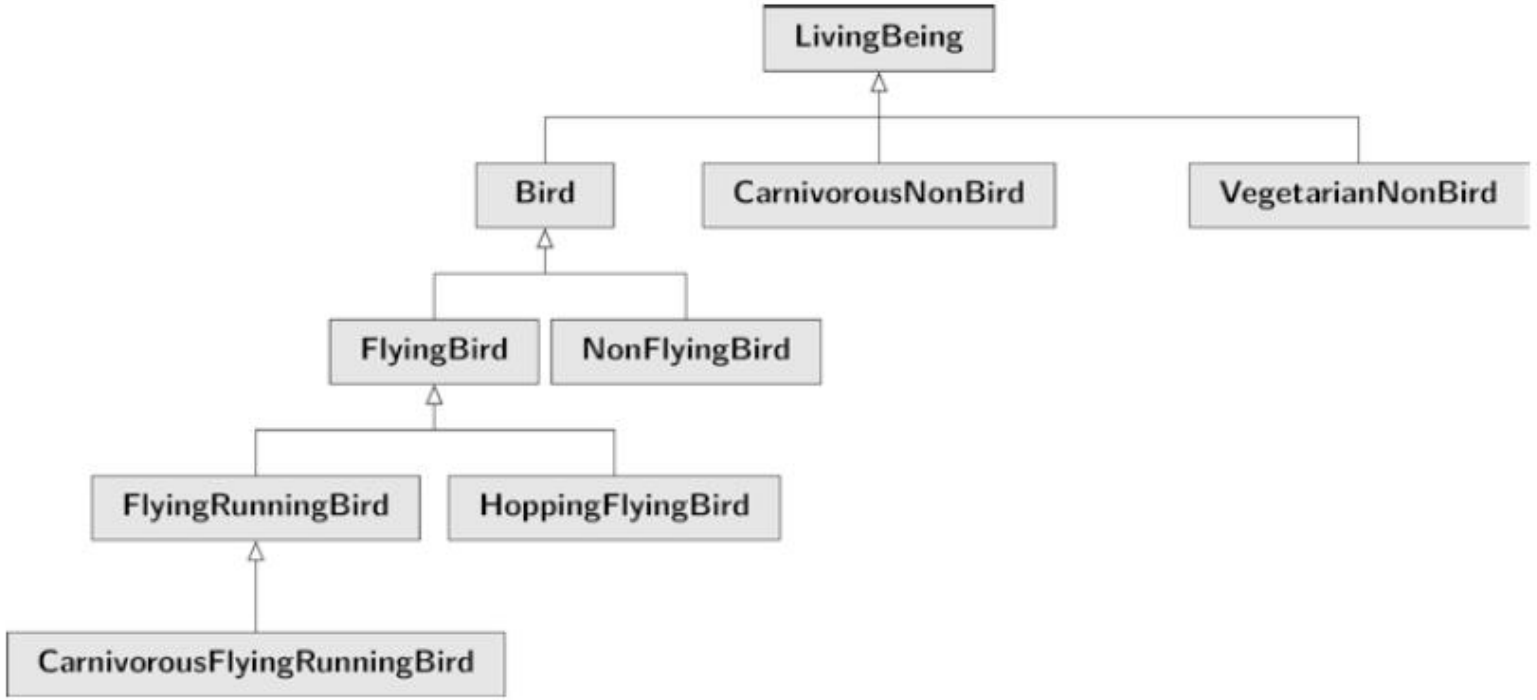
Si cela fonctionne, le canard a changé de comportement de vol dynamiquement ! Ce serait IMPOSSIBLE si l'implémentation résidait dans la classe Canard

après

# Principe de conception

*Préférez la composition à l'héritage*

# Exemple 2


LivingBeing
Bird
CarnivorousNonBird
VegetarianNonBird
FlyingBird
NonFlyingBird
FlyingRunningBird
HoppingFlyingBird
CarnivorousFlyingRunningBird

# Questions

**Est-ce que je dois toujours implémenter mon application d'abord, voir les éléments qui changent, puis revenir en arrière pour séparer et encapsuler ces éléments ?**

# Questions

**Est-ce qu’on devrait aussi transformer Canard en interface ?**

# Questions

**Cela fait vraiment bizarre d'avoir une classe qui n'est qu'un comportement. Est-ce que les classes ne sont pas censées représenter des *entités ? Est-ce qu'elles ne doivent pas avoir un état ET un comportement ?***

# Questions

Un appeau est un instrument que les chasseurs emploient pour imiter les appels (cancanements) des canards. Comment implémenteriez-vous un appeau *sans* hériter de la classe Canard ?

# Le pattern Stratégie

Il 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.

# Problème de conception

Réorganiser les classes.

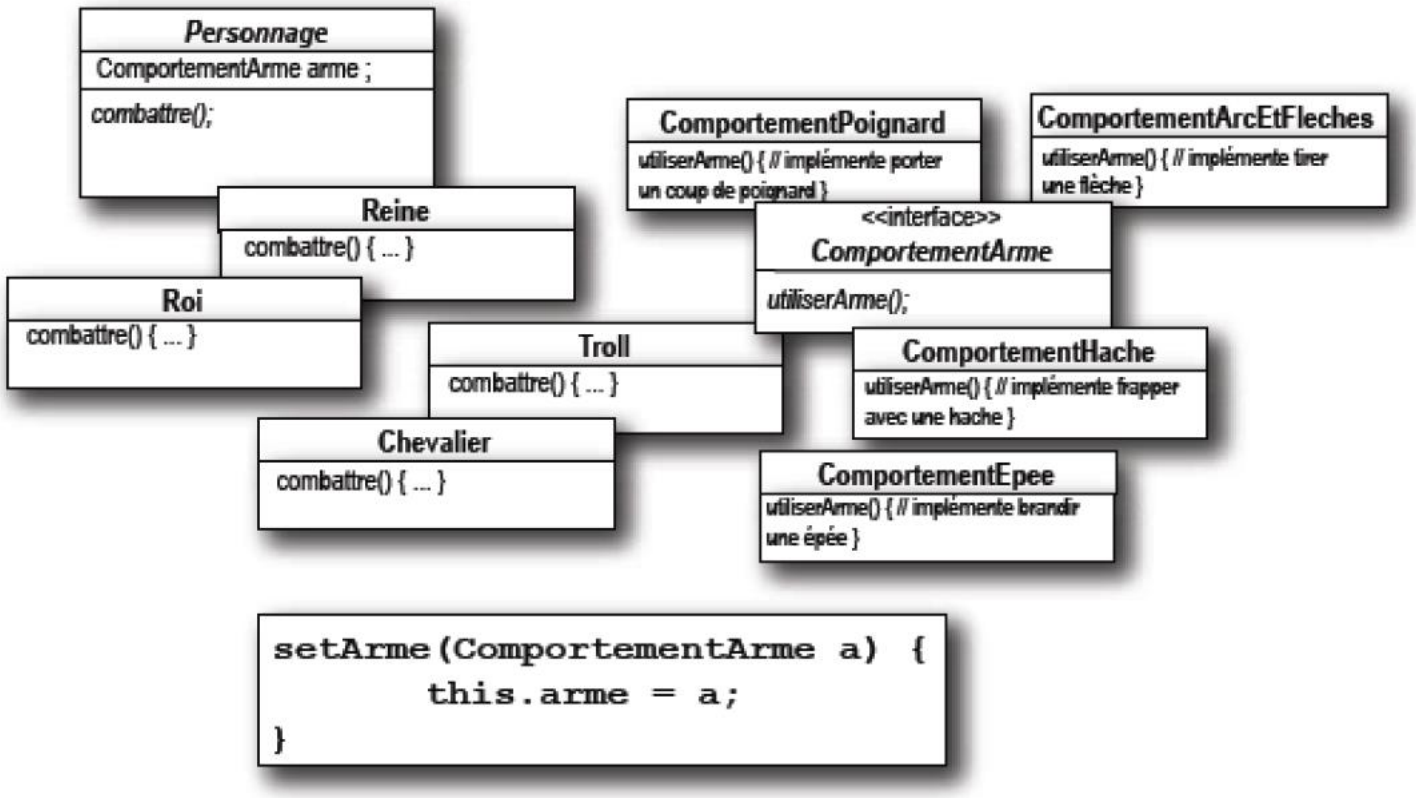
Identifier une classe abstraite, une interface et huit classes ordinaires.

Tracer des flèches entre les classes (Vu dans le cours d UML).

Tracez un type de flèche pour l'héritage (« extends »).

Tracez un type de flèche pour l'interface (« implements »).

Tracez un type de flèche pour «A-UN».



# Questions

**Si les design patterns sont tellement géniaux, pourquoi quelqu'un ne les a-t-il pas transformés en bibliothèque pour que je n'aie plus rien à faire ?**

# Questions

**Les bibliothèques et les frameworks ne sont donc pas des design patterns ?**

# Design patterns

| | Class-based | Object-based |
|---|---|---|
| Creational patterns (Generation patterns) | Factory method | Abstract factory |
| | | Builder |
| | | Prototype |
| | | Singleton |
| Structural patterns (Structural pattern) | Adapter | Adapter |
| | | Bridge |
| | | Composite |
| | | Decorator |
| | | Facade |
| | | Flyweight |
| | | Proxy |
| Behavioral patterns (Behavior pattern) | Interpreter | Chain of responsibility |
| | Template method | Command |
| | | Iterator |
| | | Mediator |
| | | Memento |
| | | Observer |
| | | State |
| | | Strategy |
| | | Visitor |