Définitions Design patterns

- Un Design pattern décrit à la fois
  - Un problème qui se produit très fréquemment, dans un environnement,
  - et l'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois.

- Permet de décrire avec succès des types de solutions récurrentes à des problèmes communs dans des types de situations

Définitions Design patterns

- Les design patterns offrent
  - Une documentation d'une expérience éprouvée de conception
  - Une identification et spécification d 'abstractions qui sont au dessus du niveau des simples classes et instances
  - Un vocabulaire commun et aide à la compréhension de principes de conception
  - Un moyen de documentation de logiciels
  - Une Aide à la construction de logiciels complexes et hétérogènes, répondant à des propriétés précises.

Catégories de Design Patterns

- Création
  - Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
  - Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects
  - Exemples : Abstract Factory, Builder, Prototype, Singleton
- Structure
  - Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
  - Exemples : Adapter(objet), Composite, Bridge, Decorator, Facade, Proxy
- Comportement
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets
  - Exemples : Strategy, Observer, Iterator, Mediator , Visitor, State

Portée des Design Patterns

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage
- Portée d'Instance (Objet)
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition

Design Patterns du GoF (Gang of Four) (Gamma, Helm, Johnson, Vlissides)

		Catégorie		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory Method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Présentation d'un Design Pattern

- Nom du pattern
  - utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux
- Problème
  - description des conditions d 'applications. Explication du problème et de son contexte
- Solution
  - description des éléments (objets, relations, responsabilités, collaboration)
  - permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, …
  - vision statique ET dynamique de la solution
- Conséquences
  - description des résultats (effets induits) de l 'application du pattern sur le système (effets positifs ET négatifs)

QUELQUES RAPPELS

med@youssfi.net

Exigences d'un projet informatique

- Exigences fonctionnelles:
  - Une application est créée pour répondre , tout d'abord, aux besoins fonctionnels des entreprises.
  - Processus métier de l'entreprise
- Exigences Techniques :
  - Les performances:
  - La maintenance:
  - Sécurité
  - Portabilité
  - Distribution
  - Architectures orientées services
  - Capacité de fournir le service à différents type de clients (Desk TOP, Mobile, SMS, http…)
  - …..
- Exigence financières

Constat

- Il est très difficile de développer un système logiciel qui respecte ces exigences sans utiliser l'expérience des autres :
  - Réutiliser des modèles de conceptions (Design Patterns)
  - Bâtir les applications sur des architectures existantes:
    - Architecture JEE
    - Architecture Dot Net
    - …
  - Ces architectures offrent :
    - Des Frameworks qui permettent de satisfaire les exigences techniques:
      - Framework pour l'Inversion de contrôle
        - Gérer le cycle de vie des composants de l'application
        - Séparer le code métier du code technique
      - Framework ORM (Mapping Objet Relationnel)
      - Framework MVC WEB
      - ….
    - Middlewares pour faire communiquer les composants distribués de l'application

Concepts fondamentaux de l'orienté Objet

- Classe
- Objet
- Héritage
- Encapsulation
- Polymorphisme

med@youssfi.net

Pourquoi l'approche Orientée Objet a été créée ?

- Principalement :
  - Pour faciliter la réutilisation de l'expérience des autres.
    - Instancier des classes existantes (Composition)
    - Créer de nouvelles classes dérivées (Héritage)
    - Réutilisation des Frameworks
  - Créer des applications faciles à maintenir :
    - Applications fermées à la modification et ouvertes à l'extension
  - Créer des applications performantes
  - Créer des applications sécurisées
  - Créer des applications distribuées
  - Séparer les différents aspects d'une application
    - Aspects métiers (fonctionnels)
    - Aspect Présentation (Web, Mobile, Desktop, ….)
    - Aspects Techniques
      - ORM, Gestion des transaction, Sécurité, Montée en charge
      - Distribution

HÉRITAGE ET COMPOSITION

med@youssfi.net

Héritage et composition

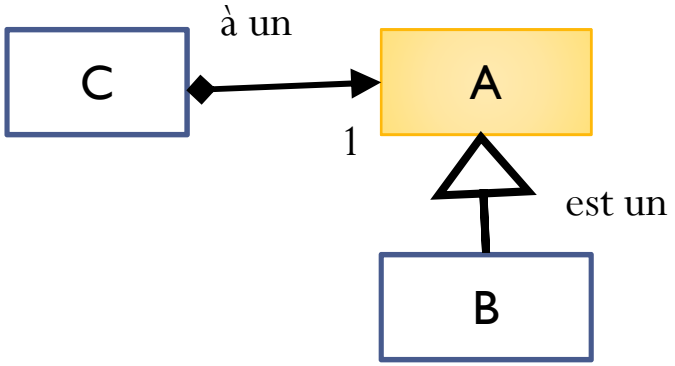
- Dans la programmation orientée objet, l'héritage et la composition sont deux moyens qui permettent la réutilisation des classes
- L'héritage traduit le terme « Est un » ou « Une sorte de »
- La composition traduit le terme « A un » ou « A plusieurs ».
- Voyons à partir d'un exemple entre l'héritage et la composition de type « A un ».

Exemple Héritage et composition

```
public class A {
  int v1=2;
  int v2=3;
  public int meth1(){
       return(v1+v2);
  }
}
```

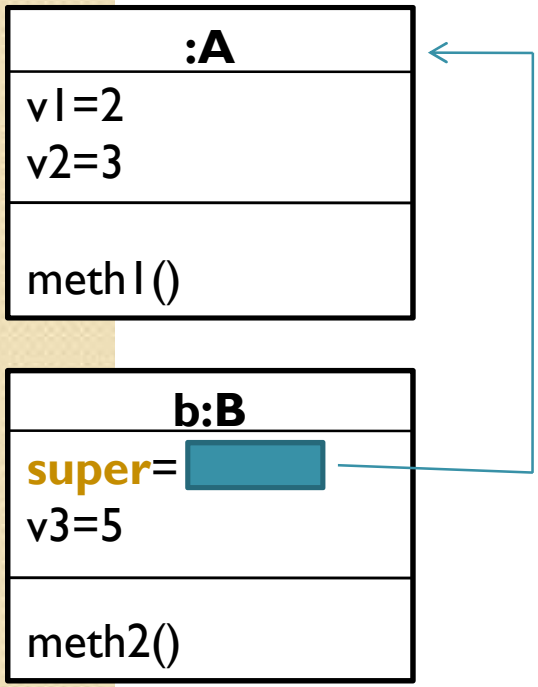
```
public class B extends A {
  int v3=5;
  void meth2(){
System.out.print(super.meth1()*v3);
  }
}
```

```
public class C {
 int v3=5;
 A a=new A();
void meth2(){
 System.out.print(a.meth1()*v3);
 }
}
```

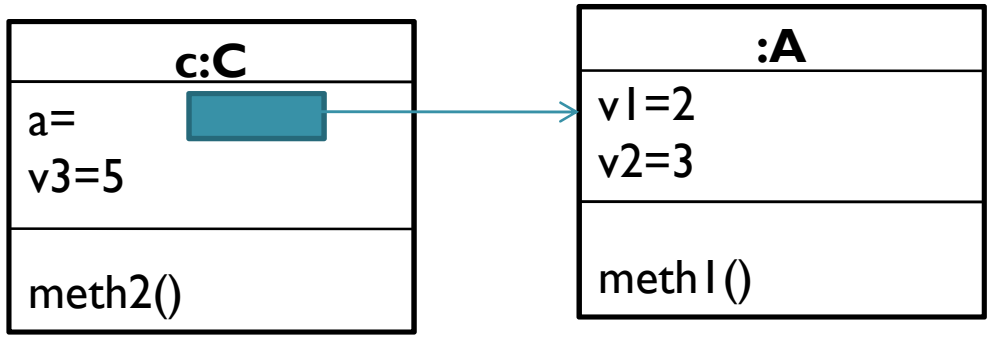


Héritage et composition

- `B b=new B();`
- `b.meth2();`



- `C c=new C();`
- `c.meth2();`



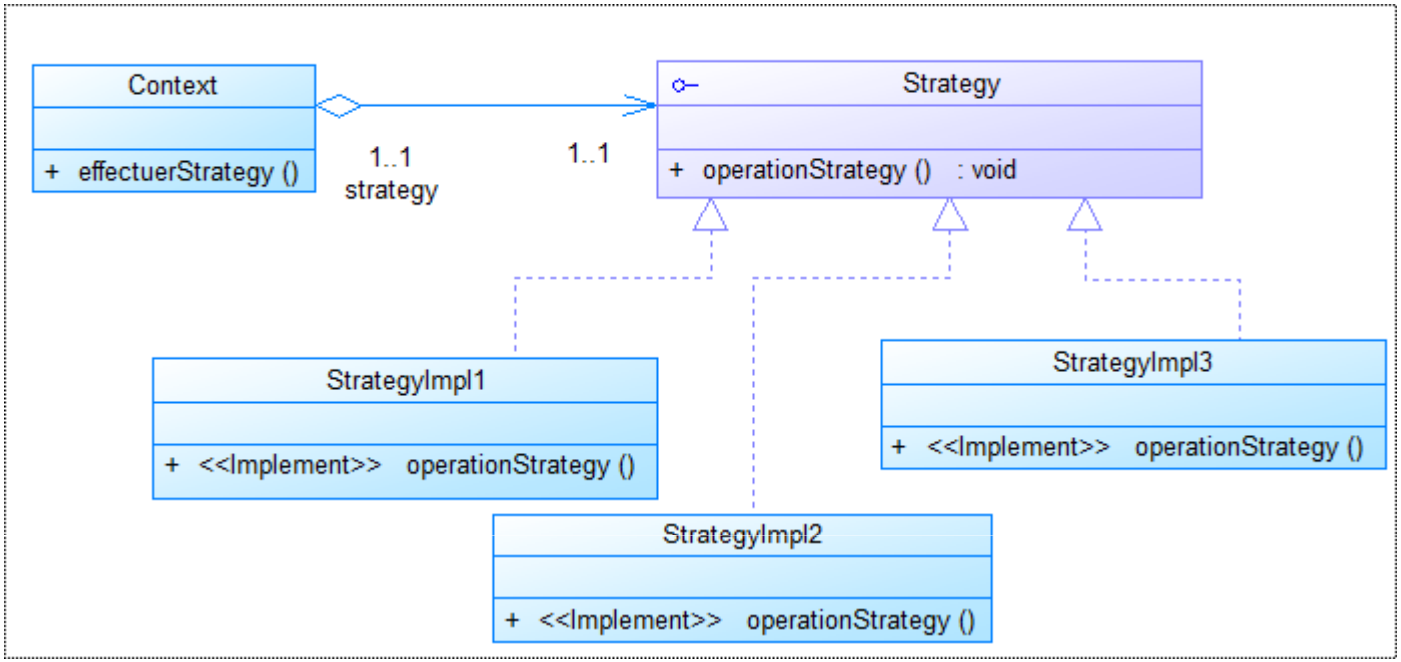
Portée des Design Patterns

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage
- Portée d'Instance (Objet)
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition

Pattern Strategy

- Catégorie : Comportement
- Objectif s:
  - Définir une famille d'algorithmes, et encapsuler chacun et les rendre interchangeables tout en assurant que chaque algorithme puisse évoluer indépendamment des clients qui l'utilisent
- Raison d'utilisation :
  - Un objet doit pouvoir faire varier une partie de son algorithme dynamiquement.
- Résultat :
  - Le Design Pattern permet d'isoler les algorithmes appartenant à une même famille d'algorithmes.

Diagramme de classes



• On crée donc une **interface** de base, appelée ici « ***Strategy »*** et on y ajoute une méthode qui sera la méthode qui applique notre stratégie.
• Il suffit alors de créer maintenant des classes concrètes qui implémentent cette interface « **StrategyImpl** » et qui donc redéfinisse la méthode de stratégie.
• A un instant donné, La classe « **Context** » qui va utiliser la stratégie compose une instance de l'une des implémentation de Strategy.

Implémentation Java du pattern Strategy

Interface Strategy.java

```
public interface Strategy {
   public void operaionStrategy();
}
```

Classe Context.java

```
public class Context {
  protected Strategy strategy;

  public void appliquerStrategy(){
       strategy.operaionStrategy();
  }
  public void setStrategy(Strategy strategy) {
       this.strategy = strategy;
  }
}
```

Implémentation Java du pattern Strategy

Implémentation 1 de l'interface Strategy.

```
public class StrategyImpl1 implements Strategy {
   @Override
   public void operaionStrategy() {
        System.out.println("Application de Strategy 1");
   }
}
```

Implémentation 2 de l'interface Strategy.

```
public class StrategyImpl2 implements Strategy {
   @Override
   public void operaionStrategy() {
        System.out.println("Application de Strategy 2");
   }
}
```

Utilisation du pattern Strategy

Application.java

```
public class Application {
public static void main(String[] args) {
    Context ctx=new Context();
    System.out.println("Stratégie 1:");
    ctx.setStrategy(new StrategyImpl1());
    ctx.appliquerStrategy();
    System.out.println("Stratégie 2:");
    ctx.setStrategy(new StrategyImpl2());
    ctx.appliquerStrategy();
    System.out.println("Stratégie 3:");
    ctx.setStrategy(new StrategyImpl3());
    ctx.appliquerStrategy();
}
}
```

Exécution :

```
Stratégie 1:
Application de Strategy 1
Stratégie 2:
Application de Strategy 2
Stratégie 3:
Application de Strategy 3
```

EXEMPLE D'APPLICATION

med@youssfi.net

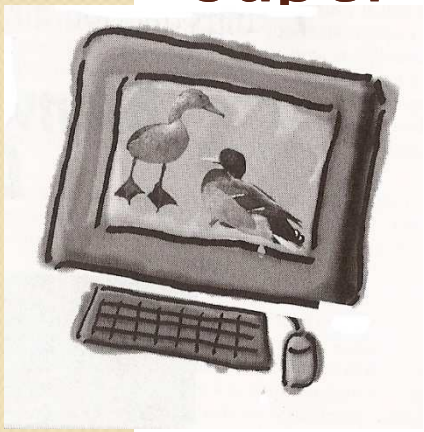
Référence


Un guide d'apprentissage
Tête la première
Design Patterns
O'REILLY

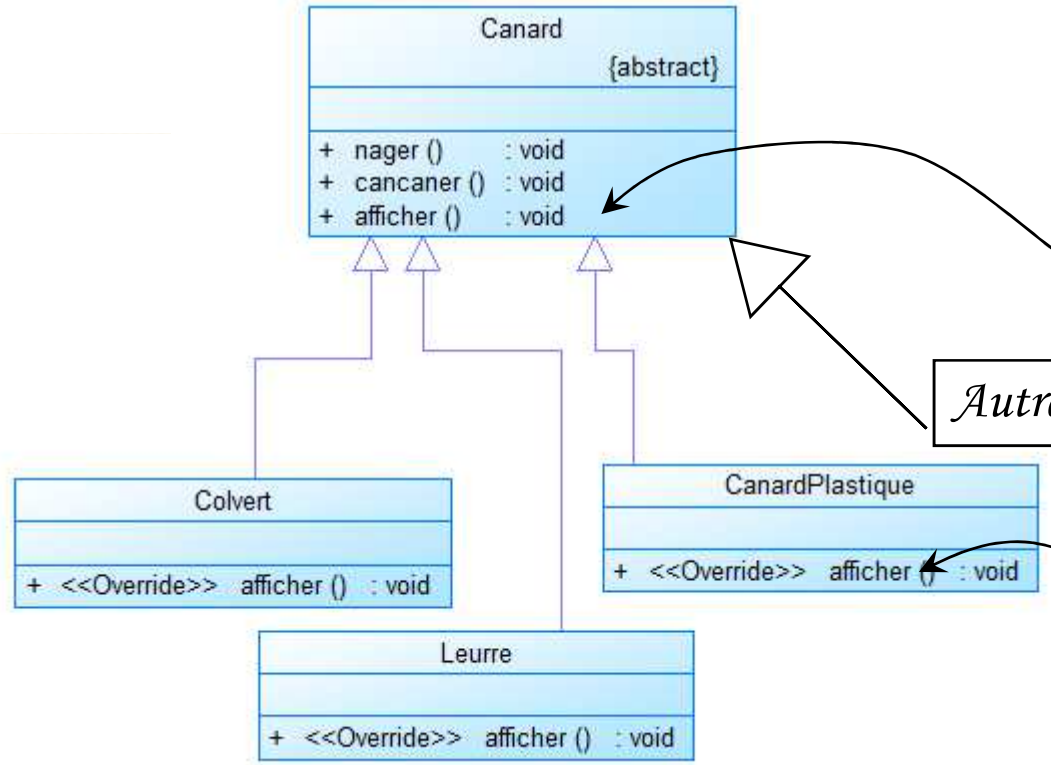
Simple Application: SuperCanard

- Joël travaille pour une société qui a rencontré un énorme succès avec un jeu de simulation de mare aux canard « 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 super classe Canard dont tous les autres types de canards héritent.

SuperCanard dans le passé



Tous les canard cancanent et nagent. La super classe gère le code de l'implémentation



Comme tous les sous types de la classes canard ont un aspect différent, la méthode afficher est abstraite

Autres types de canards

Chaque sous classe de canard a la charge d'implémenter le comportement de la méthode afficher() pour la façon dont il apparaîtra à l'écran

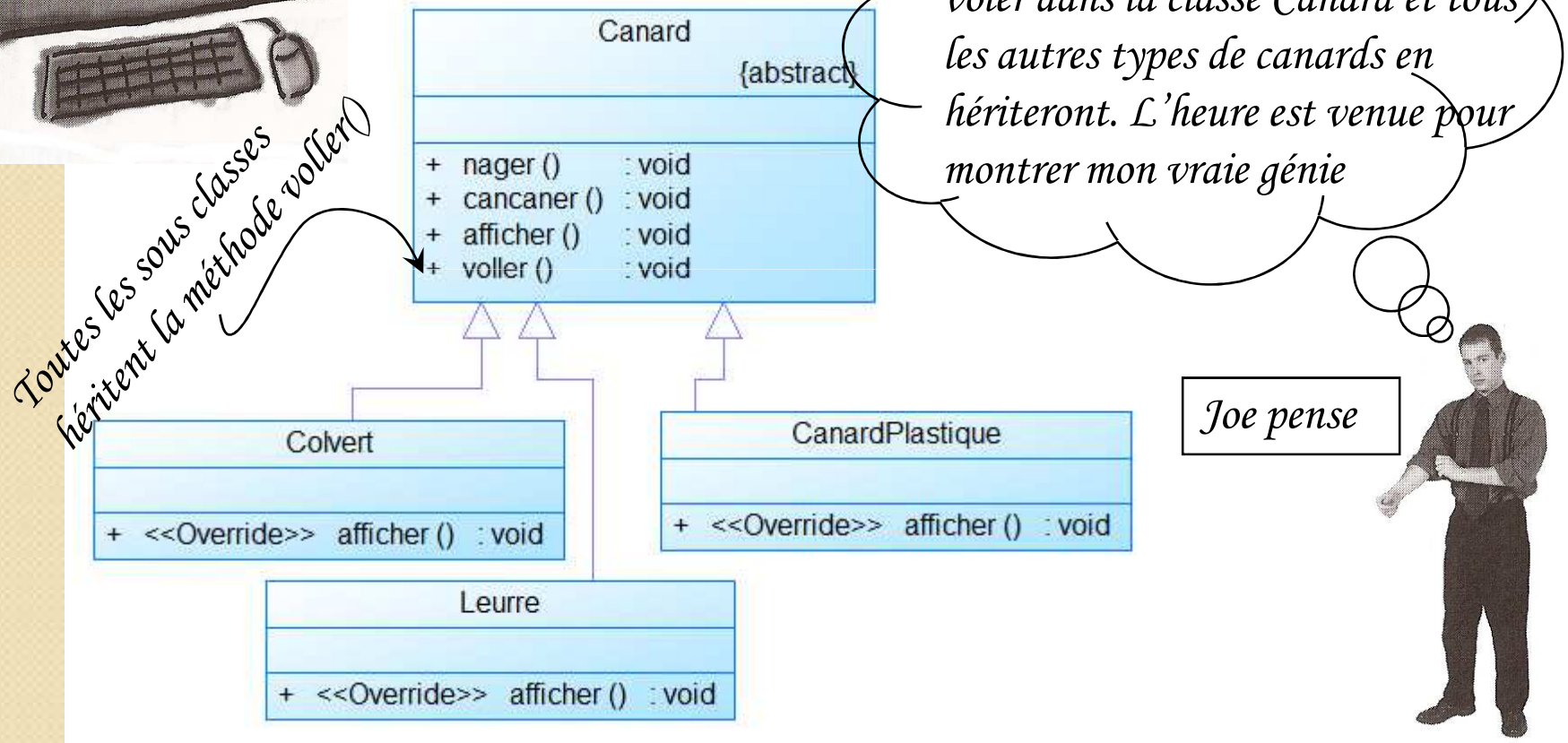
Super canard

- La société a subi de plus en plus la pression de la part de la concurrence.
- A l'issue d'une semaine d'un séminaire résidentiel consacré au brainstorming, les dirigeants ont pensé qu'il était temps de lancer une grande innovation.
- Il leur faut, maintenant, quelque chose de réellement impressionnant à présenter à la réunion des actionnaires qui aura lieu la *semaine prochaine.*

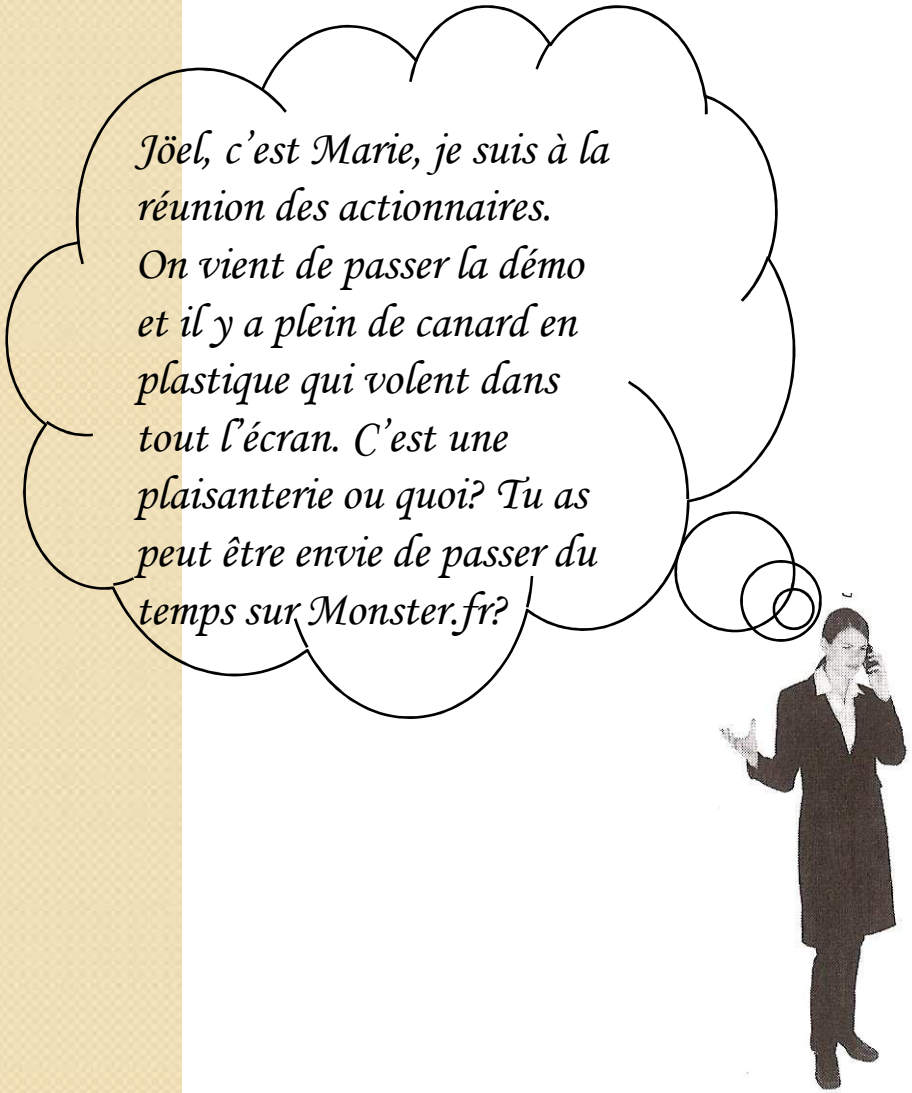
Maintenant, nous voulons que les canards **VOLENT**

- Les dirigeant ont décidé que les canards volant étaient exactement ce qu'il fallait pour battre la concurrence.
- Naturellement, le responsable de Joël a dit que celui-ci n'aurait aucun problème pour bricoler quelque chose en une semaine.
- « *Après tout* », a-t-il dit, « *Jöel est un programmeur OO… ça ne doit pas être si difficile !* »

Maintenant, nous voulons que les canards VOLENT



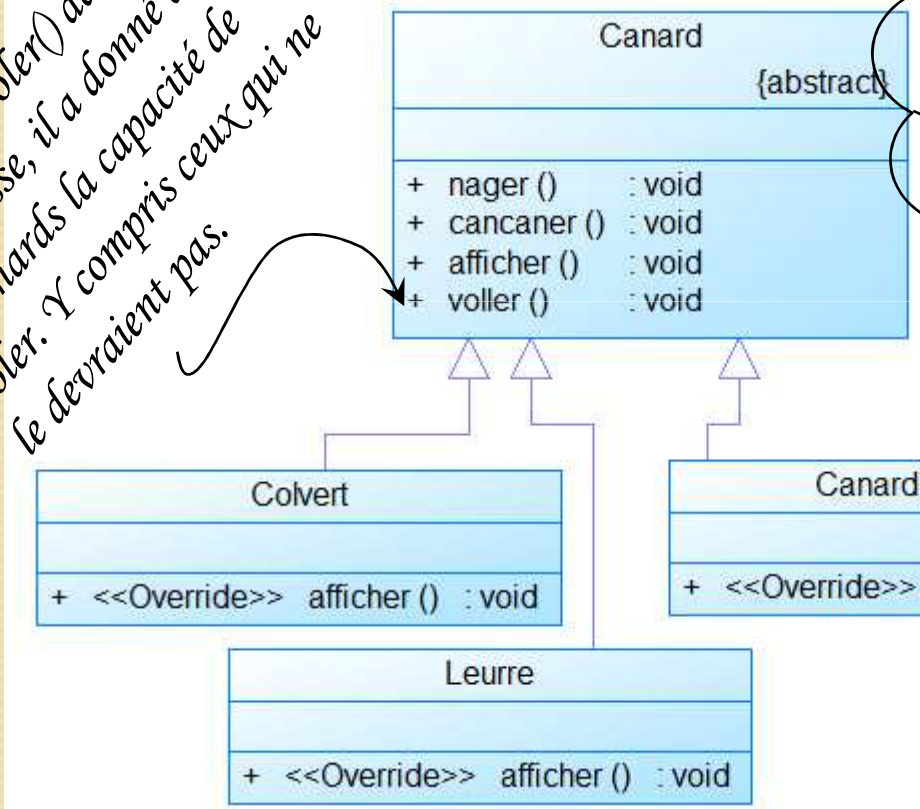
Nous somme à la réunion: Il y a quelque chose qui ne va pas



- Que s'est t-il passé?
  - Jöel a oublié que toutes les sous-classes de canard ne doivent pas voler.
  - Quand il a ajouté le nouveau comportement de la super classe Canard, il a également ajouté un comportement non approprié à certaines de ses sous-classes;
  - Maintenant, il a des objets volant inanimés dans son programme SuperCanard

Il y a quelque chose qui ne va pas

En plaçant voler() dans la super classe, il a donné à tous les canards la capacité de voler. Y compris ceux qui ne le devraient pas.



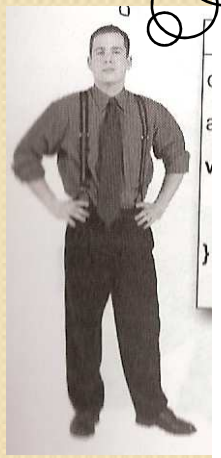
OK, On dirait qu'il y a un léger défaut dans ma conception. Je ne vois pas pourquoi il ne peuvent pas appeler ça une « fonctionnalité ». C'est marrant…..

Que pense Joe?

Jöel réfléchit à l'héritage…

Je pourrais toujours me contenter de redéfinir la méthode voler() dans la sous-classe CanardEnPlastique, comme je l'ai fait pour cancaner()

Sauf que le problème c'est que je dois faire la même chose pour les tous les autres types de canards qui ne volent pas. En effet les canards de type Leure ne volent pas et ne cancanent pas. Je crois que cette manière de faire me poserait un problème pour la maintenance



Que pense Joe?

CanardEnPlastique
cancaner() { //Redéfinir pour couiner } **voler(){** **// Redéfinir pour ne rien faire** **}** afficher() { //Aspect d'un colvert }

Leurre
cancaner() { //Redéfinir pour ne rien faire } **voler(){** **// Redéfinir pour ne rien faire** **}** afficher() { //Aspect d'un colvert }

En plus du coté fonctionnel penser à la maintenance

- Jöel s'est rendu compte que l'héritage n'est probablement pas la réponse.
- En fait, il vient de recevoir une note annonçant que les dirigeant ont décidé de réactualiser le produit tous les six mois.
- Il sait que les spécifications vont changer en permanence et qu'il sera peut être amené à redéfinir voller() et cancaner() pour toute sous classes de Canard qui sera ajoutée au programme.
- Il a donc besoin d'un moyen plus sain pour que seuls certains types de canard puissent voler ou cancaner.

La seule et unique constante du développement

- Quelle est la seule chose sur laquelle vous puissiez toujours compter en tant que développeur?
  - Indépendamment de l'endroit ou vous travaillez, de l'application que vous développez ou du langage dans lequel vous programmez, La seule vraie constante qui vous accompagnera toujours est:

Prenez un miroir pour voir la réponse pour ne jamais l'oublier

- Une application qui n'évolue pas meurt

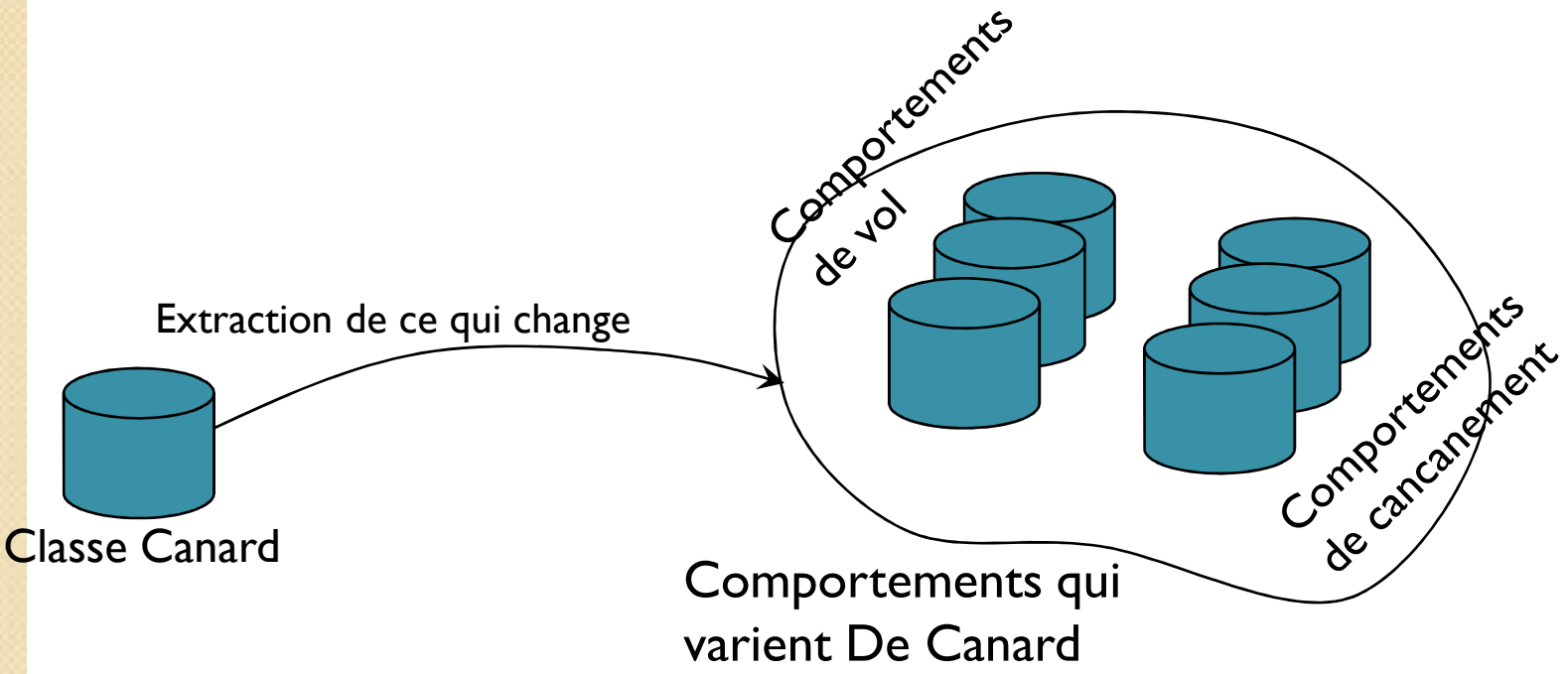
Attaquons le problème

Principe de conception:
Identifier les aspects qui changent dans votre application et séparez-les de ceux qui demeurent constants.

1er Principe de conception

Séparons ce qui change de ce qui reste identique.

- Pour l'instant, en dehors du problème de voler() et de cancaner(), la classe Canard fonctionne bien et ne contient rien qui semble changer fréquemment. Nous allons la laisser pratiquement telle quelle.
- Maintenant, pour séparer les parties qui changent de celles qui restent identiques, nous allons créer deux ensembles de classes, totalement distincts de Canard, l'un pour voler et l'autre pou cancaner.
- Chaque ensemble de classes contiendra toutes les implémentations de leur comportement respectif.



Conception des comportements de Canard

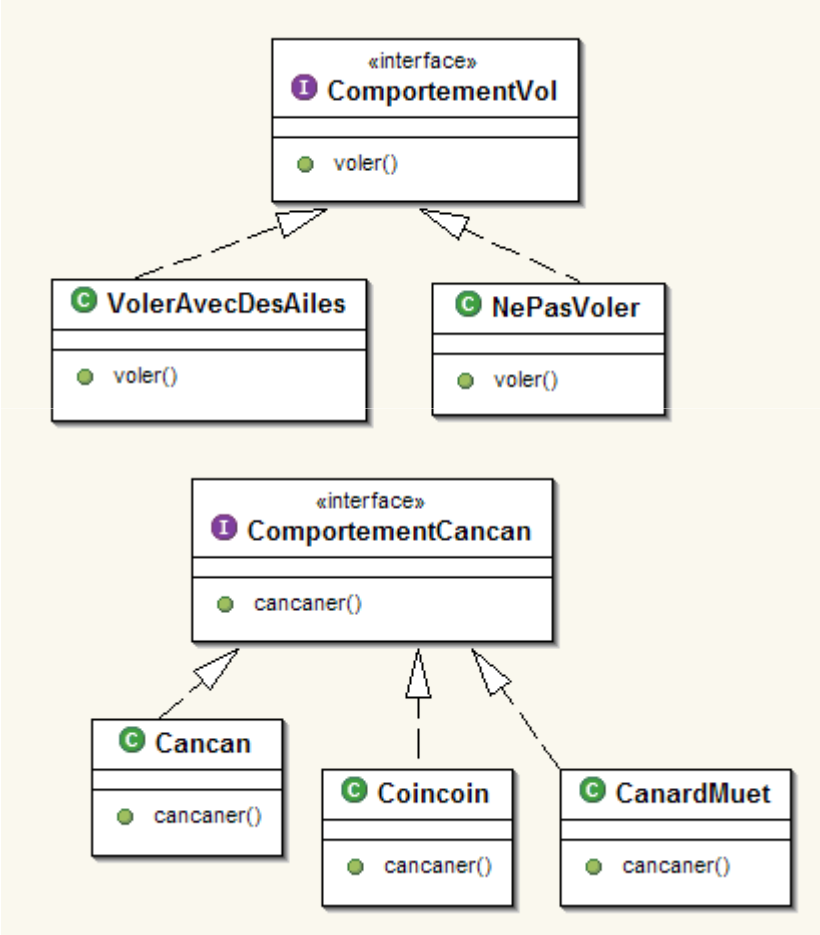
- Comment allons-nous procéder pour concevoir les classes qui implémentent les deux types de comportements?
  - Pour conserver une certaine souplesse, nous allons procéder de façon que nous puissions affecter un comportement de vol de manière dynamique à un objet Canard créé.
  - Avec ces objectifs en tête, voyons notre deuxième principe de conception

Principe de conception:
Programmer une interface, non une implémentation

2ème Principe de conception

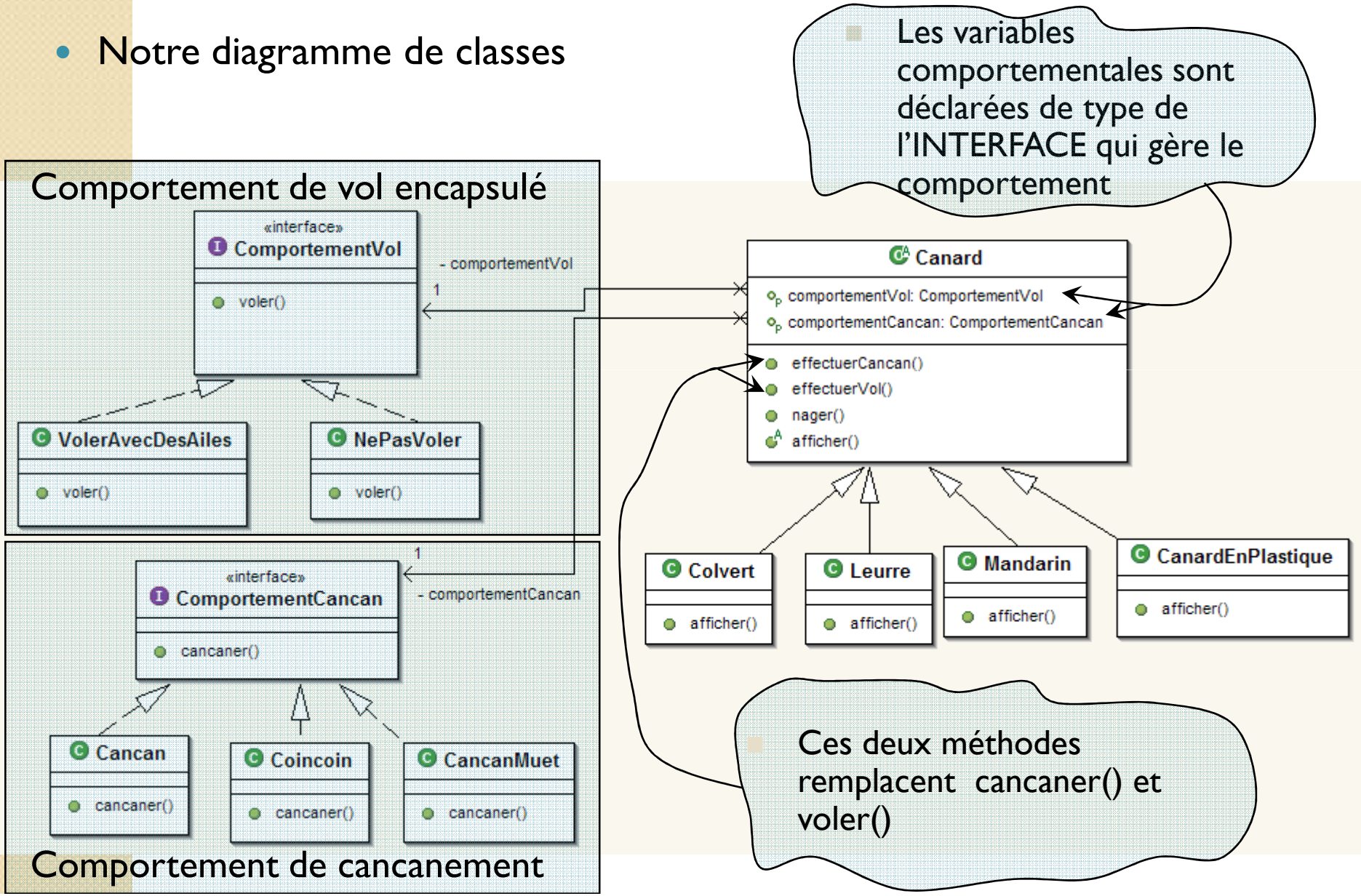
Conception des comportements de Canard

- Nous allons donc utiliser une interface pour représenter chaque comportement; par exemple, ComportementVol et ComportementCancan.
- Et chaque implémentation d'un comportement implémentera l'une de ces deux interfaces.
- Cette fois, ce n'est pas les classes Canard qui implémenteront ces interfaces, mais nous allons créer un ensemble de classes qui implémenteront ces interfaces et dont la seule raison d'être est de représenter un comportement



Intégrer les comportements des canards

- Notre diagramme de classes



Conception des comportements de Canard

- Avec cette conception, les autres types de d'objets peuvent réutiliser nos comportements de vol et de cancanement parce que ces comportements ne sont pas cachés dans nos classes Canard!
- Et nous pouvions ajouter de nouveaux comportements sans modifier aucune des classes comportementales existantes, ni toucher à aucune des classes Canard qui utilisent les comportements de vol et de canacanement.
- Nous obtenons ainsi les avantages de la réutilisation sans la surcharge qui accompagne l'héritage.

Intégrer les comportements des canards

- La clé est qu'un canard va maintenant déléguer ses comportements au lieu d'utiliser les méthodes voler et cancaner() définies dans la classe Canard (ou une sous-classe)
- Nous allons d'abord ajouter à la classe Canard deux variables d'instance nommées:
  - comportementVol
  - comportementCancan
- Ces attributs seront déclarés de type d'interface et non de type d'implémentation concrète.
- Chaque objet Canard affectera à ces variables de manière polymorphe pour référencer le type de comportement spécifique qu'il aimerait avoir à l'exécution (VolerAvecDesAiles, Coincoin, etc…)

Implémentons maintenant la méthode effectuerVol()

```
public abstract class Canard {
    ComportementVol comportementVol;
    //autres variables

public void effectuerVol() {
        comportementVol.voler();
    }
    // Autres méthodes
    }
```

- Chaque Canard à une référence à quelque à un objet qui implémente l'interface ComportementVol

- Au lieu de gérer son vol lui-même, l'objet Canard délègue ce comportement à l'objet référencé la variable d'instance comportementVol
- La variable d'instance comportementVol est déclarée du type de l'INTERFACE Comportementale.

- Rien de plus simple c'est ce pas?
- Pour voler, l'objet Canard demande à l'objet référencé par comportVol de le faire à sa place
- Dans ce niveau d'implémentation, peu importe de quel sorte d'objet s'agit-il.
- ***Une seule chose nous intéresse: il sait voler***

Suite de l'integration

- Il est maintenant temps de s'occuper de la façon dont les variables d'instances comportementVol et comportementCancan sont affectées
- Jeton un coup d'œil sur la sous-classe Colvert

```
public class Colvert extends Canard {
   public Colvert(){
        comportementCancan=new Cancan();
        comportementVol=new VolerAvecDesAilles();
   }
   public void afficher(){
System.out.println("Je suis un vrai Colvert");
   }
}
```

- En créant un Colvert, la responsabilité de cancanement est déléguée à un objet de type Cancan
- Quand on fait appel à la méthode effectuerCancan() c'est la méthode cancaner() de l'objet Cancan qui va s'exécuter

- Souvenez-vous que les variables compVol et compCancan sont héritée de la classe Canard

- Le comportement de vol sera délégué à un objet de type VolerAvecDesAilles

Suite de l'intégration

- Le cancan de Colvert est un vrai cancan, pas un coincoin, ni un cancan muet.
- Quand un Colvert est instancié, son constructeur initialise sa variable d'instance comportementCancan héritée en lui affectant une nouvelle instance de type Cancan (une classe d'implémentation concrète de l'interface ComportementCancan)
- Il en va de même pour comportementVol. Le constructeur de Colvert ou de Mandarin initialise la variable d'instance comportementVol avec une instance du comportement VolerAvecDesAilles (Une classe d'implémentation concrète de l'interface ComportementVol.

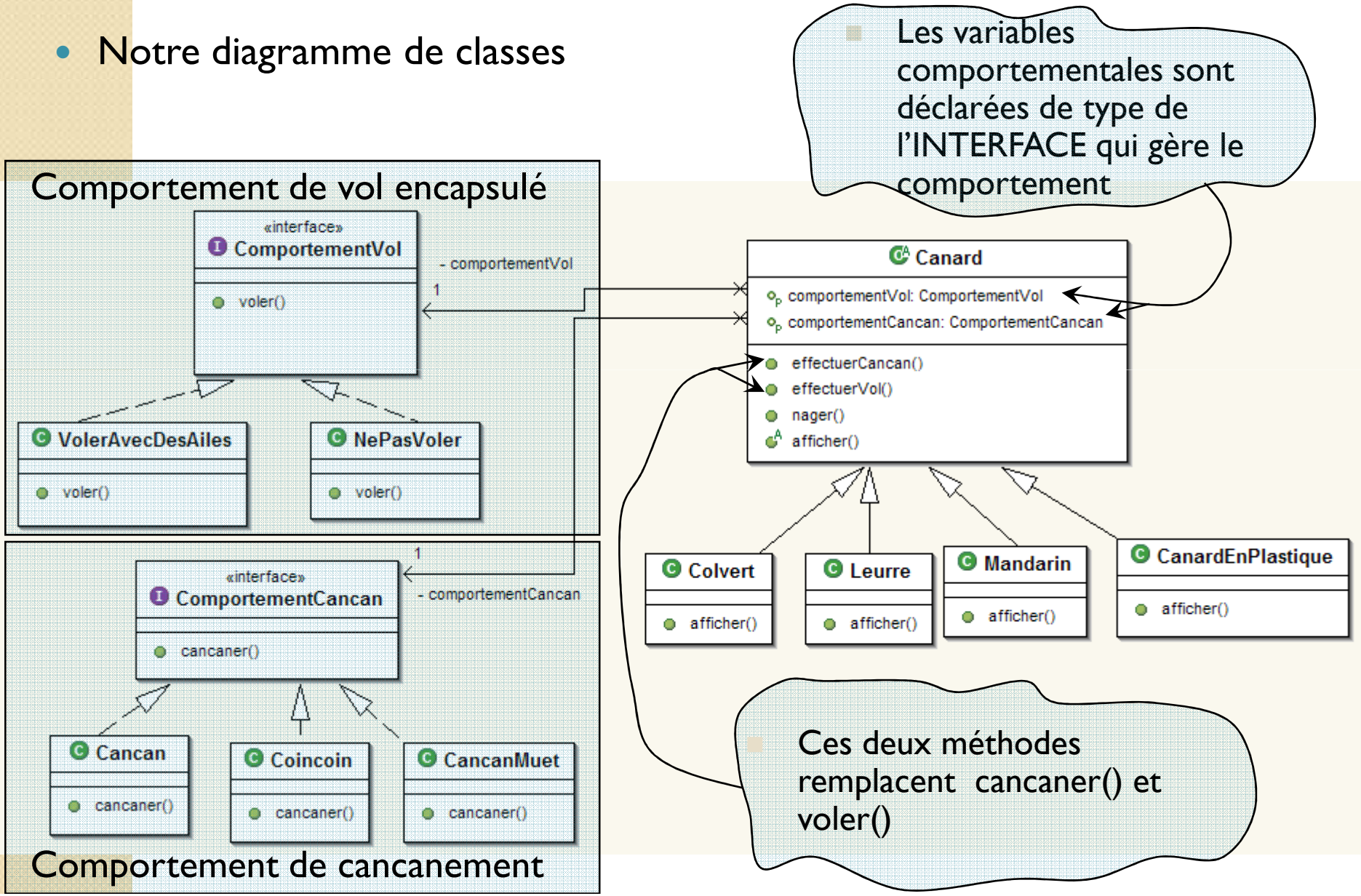
Les développeurs critiquent

Attends une seconde. Tu n'as pas dit qu'on ne doit pas programmer une implémentation? Mais , qu'est ce qu'on fait dans le constructeur de Colvert? On a créé une instance d'une classe d'implémentation concrète de Cancan: ***new Cancan()***

- Bien vue. C'est exactement ce que nous faisons…. pour l'instant.
- Plus tard, nous aurons dans notre boite à outils de conception d'autres patterns qui nous permettent d'y remédier.
- Remarquez quand même que si nous affectons bien les comportements aux classes concrètes (en instanciant une classe comportementale comme Cancan ou VolerAvecDesAilles et en affectant l'instance à notre variable de référence comportementale), nous pourrions facilement cela au moment de l'exécution en faisant appel aux setters:
  - **setComportementVol(ComportementVol cc)**
  - **setComportementCancan(ComportementCancan cc)**
- Nous avons toujours beaucoup de souplesse, mais la façon d'affecter les variables n'est pas très satisfaisante. z

Intégrer les comportements des canards

- Notre diagramme de classes



Implémenter les comportements de vol

- Interface ComportementVol.java

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

- Classe VolerAvecDesAilles.java

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

- Classe NePasVoler.java

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

Implémenter les comportements de Cancanement

- Interface ComportementCancan.java

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

- Classe Cancan.java

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

- Classe coincoin.java

```
public class CoinCoin implements ComportementCancan {
   public void cancaner() {
        System.out.println("coin coin coin!");
   }}
```

- Classe CancanMuet.java

```
public class CancanMuet implements ComportementCancan {
   public void cancaner() {
        System.out.println("Silence!");
   }}
```

Intégration des comportements aux canards

Implémentation de Canard.java

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

   public void effectuerCancan() {
        comportementCancan.cancaner();
   }
   public void effectuerVol() {
        comportementVol.voler();
   }
   public void setComportementVol(ComportementVol compV) {
        this.comportementVol = compV;
   }
   public void setComportementCancan(ComportementCancan compC) {
        this.comportementCancan = compC;
   }
   public void nager() {
        System.out.println("Je nage en flottant");
   }
   public abstract void afficher();
}
```

Intégration des comportements aux canards

Implémentation des sous classes : Colvert.java

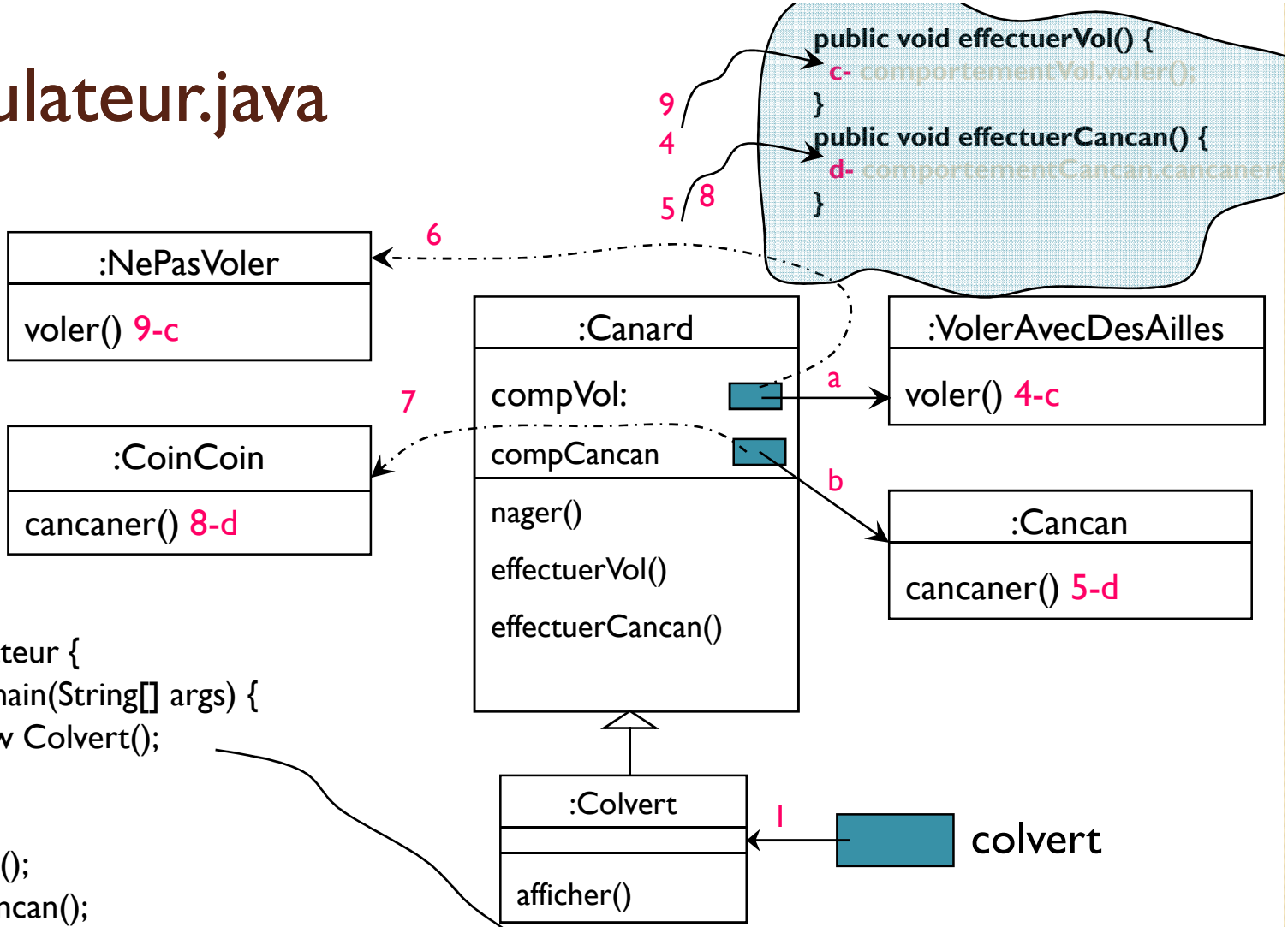
```
public class Colvert extends Canard {
   public Colvert(){
        comportementCancan=new Cancan();
        comportementVol=new VolerAvecDesAilles();
   }
   public void afficher(){
        System.out.println("Je suis un vrai Colvert");
   }
}
```

CanardEnPlastique.java

```
public class CanardEnPlastique extends Canard {
   public CanardEnPlastique(){
        comportementCancan=new CancanMuet();
        comportementVol=new NePasVoler();
   }
   public void afficher(){
        System.out.println("Je suis un canard en plastique");
   }
}
```

MiniSimulateur.java

```
C:\>java MiniSimulateur
Je suis un vrai Colvert
Je vole !
can can can
coin coin coin!
Je ne sais pas voler!
```




```
public class MiniSimulateur {
    public static void main(String[] args) {
1- Canard colvert=new Colvert();
2- colvert.nager();
3- colvert.afficher();
4- colvert.effectuerVol();
5- colvert.effectuerCancan();
6- colvert.setComportementCancan(new CoinCoin());
7- colvert.setComportementVol(new NePasVoler());
8- colvert.effectuerCancan();
9- colvert.effectuerVol();
    }
}
```

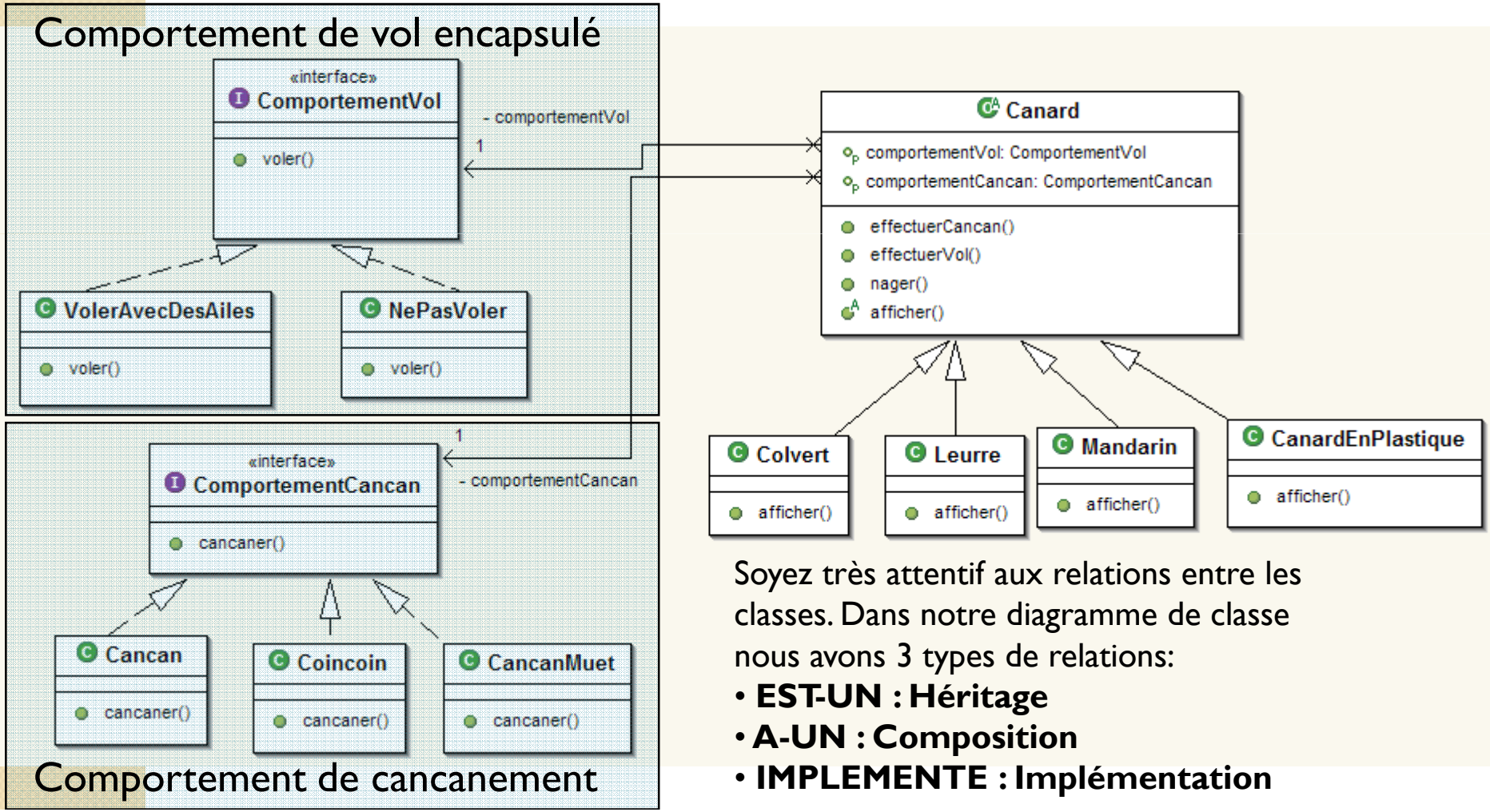
Constructeur de Colvert :

```
public Colvert(){
a- comportementCancan=new Cancan();
b- comportementVol=new VolerAvecDesAilles();
}
```

Modifier les comportements de cancan et de vol dynamiquement

Vue d'ensemble des comportements encapsulés

- Notre diagramme de classes



Soyez très attentif aux relations entre les classes. Dans notre diagramme de classe nous avons 3 types de relations:

- **EST-UN : Héritage**
- **A-UN : Composition**
- **IMPLEMENTE : Implémentation**

A-UN peut être préférable à EST-UN

- La relation A-UN est une relation intéressante :
  - Chaque Canard a un comportementVol et un comportementCancan auxquels il délègue le vol est le cancanement.
  - Lorsque vous avez deux classes de la sorte, vous utilisez le composition.
  - Au lieu d'hériter leur comportement, les canards l'obtiennent en étant composés avec le bon objet comportemental.
  - Cette technique est importante; en fait, nous avons appliqué notre troisième principe de conception:

Principe de conception:
Préférez la composition à l'héritage

3ème Principe de conception

- Utiliser la composition procure beaucoup de souplesse.
- La composition permet d'encapsuler un ensemble d'algorithmes dans leur propre ensemble de classes
- Et surtout, vous pouvez modifier le comportement au moment de l'exécution tant que l'objet avec lequel vous composez implémente la bonne interface comportementale.

Félicitation pour votre premier design pattern

- Vous venez d'appliquez votre premier design pattern appelé « STRATEGIE »
- Oui, vous avez appliqué le pattern Stratégie, pour revoir la conception de SuperCanard.
- Grâce à ce pattern, le simulateur est prêt à recevoir toutes les modifications qui pourraient provenir de la prochaine séminaire aux Baléares.
- Nous n'avons pas pris le court chemin pour l'appliquer, mais en voici la définition formelle

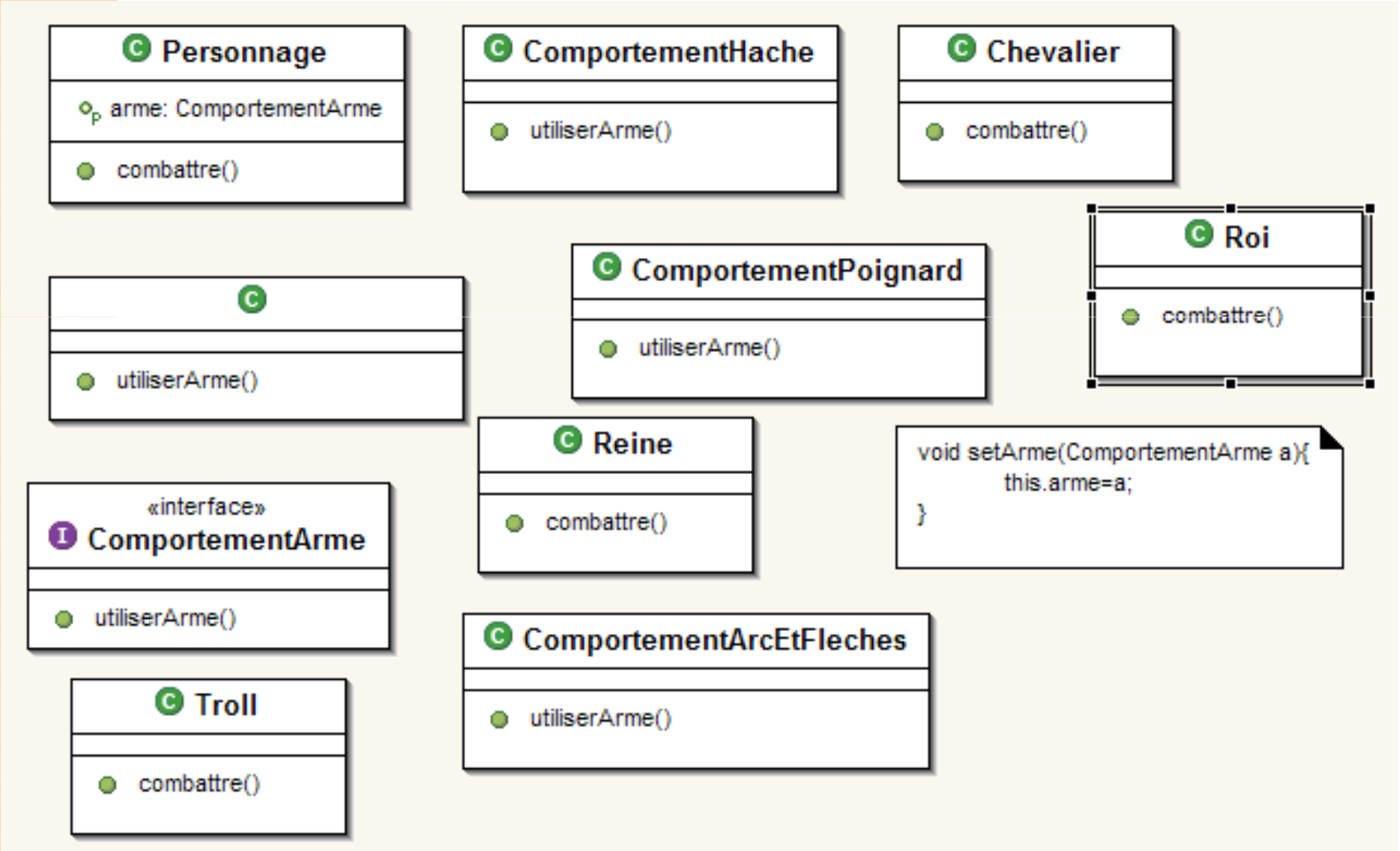
Le pattern **Stratégie** définie une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables.

Stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilise.

Problème de conception

- Vous trouverez ci-dessous un ensemble de classes et d'interfaces pour un jeu d'aventure. Elles sont toutes mélangées. Il y a des classes pour les personnages et des classes correspondants aux comportements aux armes que les personnages peuvent utiliser. Chaque personnage ne peut faire usage que d'une seule arme à la fois, mais il peut en changer à tout moment en cours du jeu. Votre tache consiste à:
  - Réorganiser les classes.
  - Identifier une classe abstraite, une interface et 8 classes concrètes
  - Définir les associations entre les classes et l'interface.
    - Respecter le type de flèche de l'héritage. (extends)
    - Respecter le type de flèche de l'implémentation.(implements)
    - Respecter le type de flèche pour la composition .(A-UN)
  - Placer le méthode setArme() dans la bonne classe

Problème de conception



Exercice d'implémentation

- On considère la classe Employe (voir annexe) qui est définie par :
  - deux variables d'instance cin et salaireBrutMensuel,
  - deux constructeurs, les getters et setters
  - et une méthode calculerIGR qui retourne l'impôt général sur les revenus salariaux.
  - La méthode getSalaireNetMensuel retourne le salaire net mensuel.
- Supposant que la formule de calcul de l'IGR diffère d'un pays à l'autre.
- Au Maroc, par exemple le calcul s'effectue selon les cas suivant :
  - Si le salaire annuel est inférieur à 40000, le taux de l'IGR est : 5%
  - Si le salaire annuel est supérieur à 40000 et inférieur à 120000, le taux de l'IGR est : 20%
  - Si le salaire annuel est supérieur à 120000 le taux de l'IGR est : 42%
- En Algérie, le calcul s'effectue en utilisant un taux unique de 35%.
- Comme cette classe est destinée à être utilisée dans différent type de pays inconnus au moment du développement de cette classe,

1. Identifier les méthodes qui vont subir des changements chez le client.
2. En appliquant le pattern strategie, essayer de rendre cette classe fermée à la modification et ouverte à l'extension.
3. Créer une application de test.
4. Proposer une solution pour choisir dynamiquement l'implémentation de calcul de l'IGR.

Code de la classe Employe

```
public class Employe {
   private String cin;
   private float salaireBrutMensuel;
        public Employe(String cin, float salaireBrutMensuel) {
                this.cin = cin;
                this.salaireBrutMensuel = salaireBrutMensuel;
        }
   public float calculerIGR(){
        float salaireBrutAnuel=salaireBrutMensuel*12;
        float taux=42;
        return salaireBrutAnuel*taux/100;
   }
   public float getSalaireNetMensuel(){
        float igr=calculerIGR();
        float salaireNetAnuel=salaireBrutMensuel*12-igr;
        return salaireNetAnuel/12;
   }
  // Getters et Setters
}
```