Un guide d'apprentissage

Tête la première Design Patterns

Évitez les erreurs de couplage gênantes

Découvrez les secrets du maître des patterns

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

Voyez pourquoi vos amis se trompent au sujet du pattern Fabrication

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®

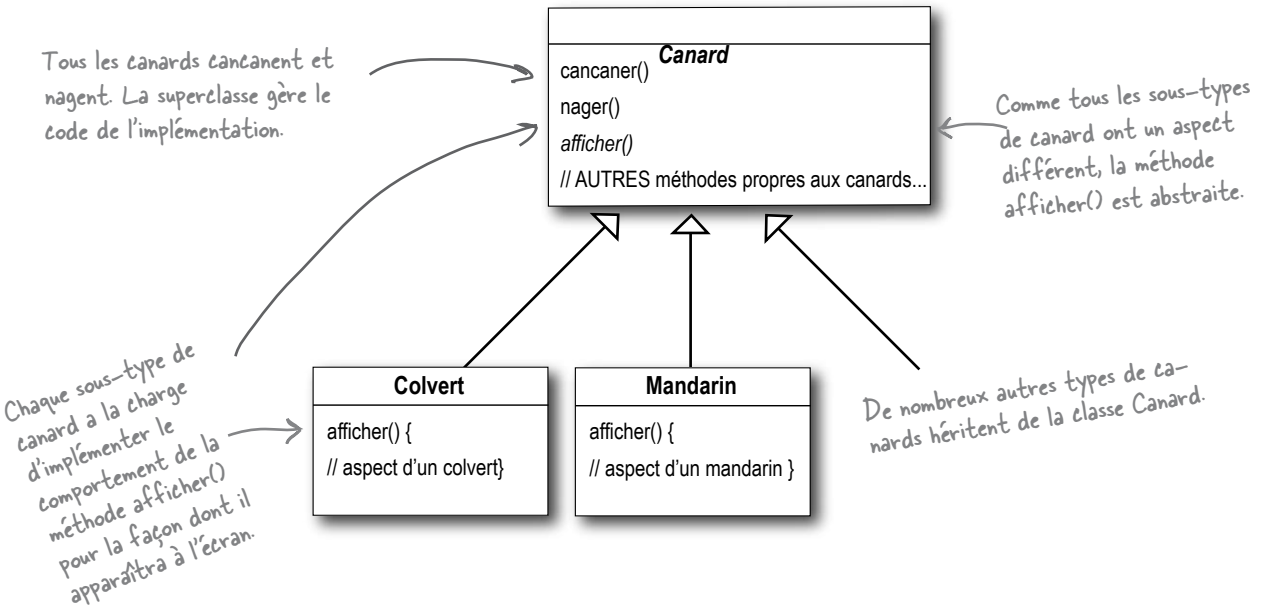
1 Introduction aux Design Patterns

Bienvenue aux Design Patterns

Quelqu'un a déjà résolu vos problèmes. Dans ce chapitre, vous allez apprendre comment (et pourquoi) 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. Nous allons 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*.

Tout a commencé par une simple application, SuperCanard

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.



L'an passé, la société a subi de plus en plus de pression de la part de la concurrence. À l'issue d'une semaine de séminaire résidentiel consacré au brainstorming et au golf, ses 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 aux Baléares *la semaine prochaine*.

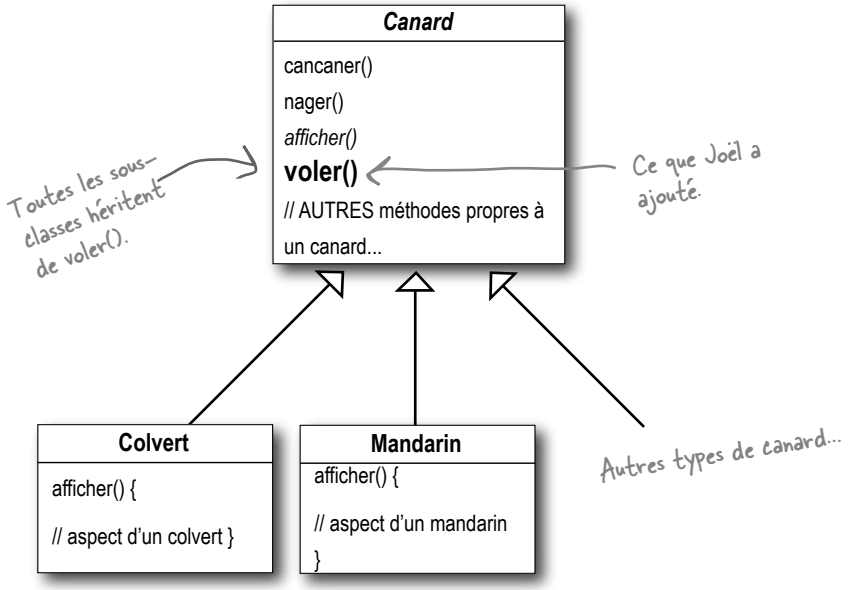
Maintenant, nous voulons que les canard VOLENT

Les dirigeants ont décidé que des canards volants étaient exactement ce qu'il fallait pour battre la concurrence à plate couture. Naturellement, le responsable de Joël leur a dit que celui-ci n'aurait aucun problème pour bricoler quelque chose en une semaine. "Après tout", a-t-il dit, "Joël est un programmeur OO... *ça ne doit pas être si difficile !*"

Ce que nous voulons.

Joe

Il suffit que j'ajoute une méthode voler() dans la classe Canard et tous les autres canards en hériteront. L'heure est venue de montrer mon vrai génie.



Il y a quelque chose qui ne va pas

Joël, je suis à la réunion des actionnaires. On vient de passer la démo et il y a plein de canards en plastique qui volent dans tout l'écran. C'est une plaisanterie ou quoi ? Tu as peut-être envie de passer du temps sur Monster.fr ?

Que s'est-il passé ?

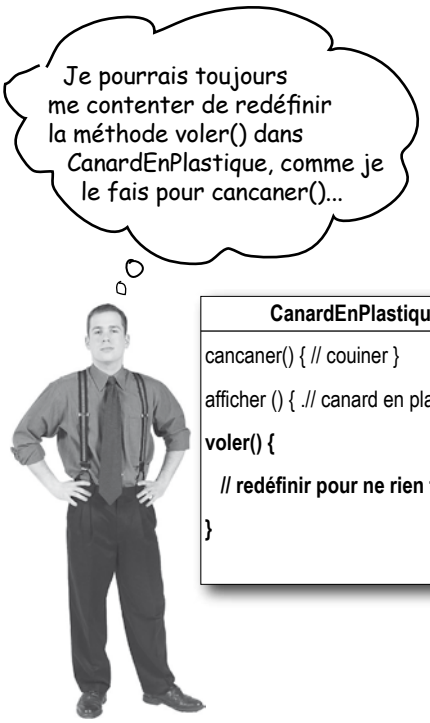
Joël a oublié que *toutes* les sous-classes de Canard ne doivent pas *voler*. Quand il a ajouté le nouveau comportement à la superclasse Canard, il a également ajouté un comportement qui n'était pas approprié à certaines de ses sous-classes. Maintenant, il a des objets volants inanimés dans son programme SuperCanard.

Une mise à jour locale du code a provoqué un effet de bord global (des canards en plastique qui volent) !

OK, on dirait qu'il y a un léger défaut dans ma conception. Je ne vois pas pourquoi ils ne peuvent pas appeler ça une "fonctionnalité". C'est plutôt marrant...

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.

Joël réfléchit à l'héritage...



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.

À vos crayons

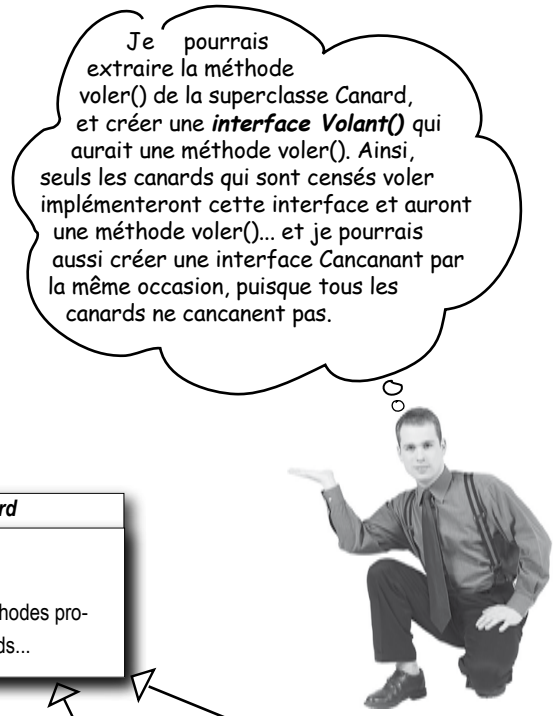
Dans la liste ci-après, quels sont les inconvénients à utiliser *l'héritage* pour définir le comportement de Canard ? (Plusieurs choix possibles.)

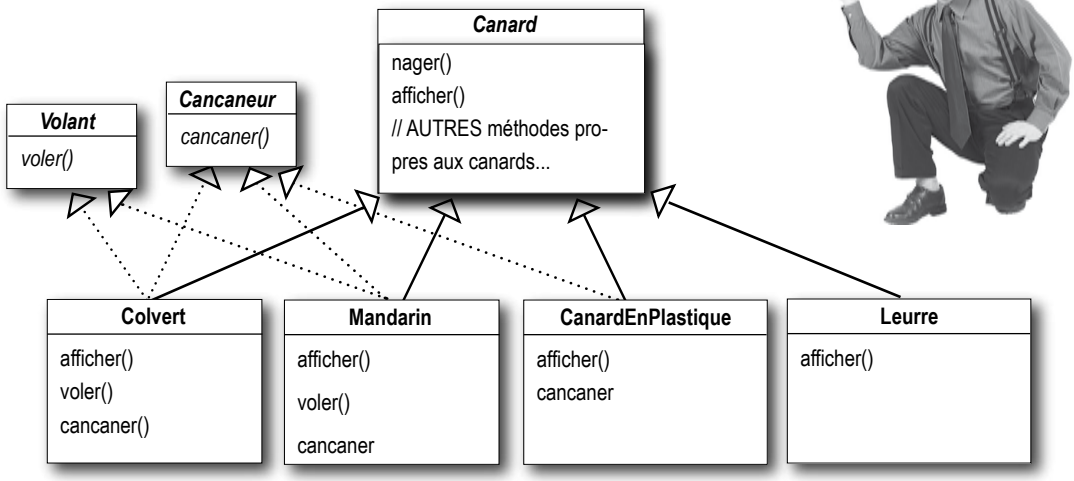
- ❑ A. Le code est dupliqué entre les sous-classes.
- ❑ B. Les changements de comportement au moment de l'exécution sont difficiles
- ❑ C. Nous ne pouvons pas avoir de canards qui dansent.
- ❑ D. Il est difficile de connaître tous les comportements des canards.
- ❑ E. Les canards ne peuvent pas voler et cancaner en même temps.
- ❑ F. Les modifications peuvent affecter involontairement d'autres canards.

Et si nous utilisions une interface ?

Joël s'est rendu compte que l'héritage n'était probablement pas la réponse : il 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 VOUS ? Que pensez-vous de cette conception ?

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 ?!

Que feriez-*vous* à la place de Joël ?

Nous savons que *toutes* les sous-classes ne doivent *pas* avoir de comportement qui permette aux canards de voler ou de cancaner. L'héritage ne constitue donc pas la bonne réponse. Mais si créer des sous-classes qui implémentent Volant et/ou Cancaneur résout une *partie* du problème (pas de canards en plastique qui se promènent malencontreusement dans l'écran), cela détruit complètement la possibilité de réutiliser le code pour ces comportements et ne fait que créer un *autre* cauchemar sur le plan de la maintenance. De plus, tous les canards qui *doivent* voler ne volent peut-être pas de la même façon...

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.*

Ne serait-ce pas merveilleux s'il y avait un moyen de construire des logiciels de sorte que les modifications aient le moins d'impact possible sur le code existant ? Cela nous permettrait de passer moins de temps à retravailler le code et plus de temps à inventer des trucs plus cools…

La seule et unique constante du développement

Bien, quelle est la seule chose sur laquelle vous puissiez toujours compter en tant que développeur ?

Indépendamment de l'endroit où vous travaillez, de l'application que vous développez ou du langage dans lequel vous programmez, quelle est la seule vraie constante qui vous accompagnera toujours ?

LE CHANGEMENT

(prenez un miroir pour voir la réponse)

Quel que soit le soin que vous ayez apporté à la conception d'une application, elle devra prendre de l'ampleur et évoluer au fil du temps. Sinon, elle *mourra*.

À vos crayons

De nombreux facteurs peuvent motiver le changement. Énumérez quelles pourraient être les raisons de modifier le code de vos applications (nous en avons listé deux pour vous aider à démarrer).

Mes clients ou mes utilisateurs décident qu'ils veulent autre chose ou qu'ils ont besoin d'une nouvelle fonctionnalité.

Mon entreprise a décidé qu'elle allait changer de système gestion de bases de données et qu'elle allait également acheter ses données chez un autre fournisseur dont le format est différent. Argh !

Attaquons le problème...

Nous savons donc que le recours à l'héritage n'a pas été un franc succès, puisque le comportement des canards ne cesse de varier d'une sous-classe à l'autre et que ce comportement n'est pas approprié à toutes les sous-classes. Les interfaces Volant et Cancaneur semblaient tout d'abord prometteuses – seuls les canards qui volent implémenteraient Volant, etc. – sauf que les interfaces Java ne contiennent pas de code : il n'y a donc pas de code réutilisable. Chaque fois que vous voulez changer un comportement, vous êtes obligé de le rechercher et de le modifier dans toutes les sous-classes dans lesquelles il est défini, en introduisant probablement quelques nouveaux bogues en cours de route !

Heureusement, il existe un principe de conception fait sur mesure pour cette situation.

Principe de conception

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

Le premier de nos nombreux principes de conception. Nous leur consacrerons plus de temps tout au long de cet ouvrage.

Autrement dit, si l'un des aspects de votre code est susceptible de changer, par exemple avec chaque nouvelle exigence, vous savez que vous êtes face à un comportement qu'il faut extraire et isoler de tout ce qui ne change pas.

Voici une autre façon de formuler ce principe : ***extraire les parties variables et les encapsuler vous permettra plus tard de les modifier ou de les augmenter sans affecter celles qui ne varient pas.***

Malgré sa simplicité, ce concept constitue la base de presque tous les design patterns. Tous les patterns fournissent un moyen de permettre à *une partie d'un système de varier indépendamment de toutes les autres.*

Cela dit, il est temps d'extraire les comportements de canard des classes Canard !

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 !

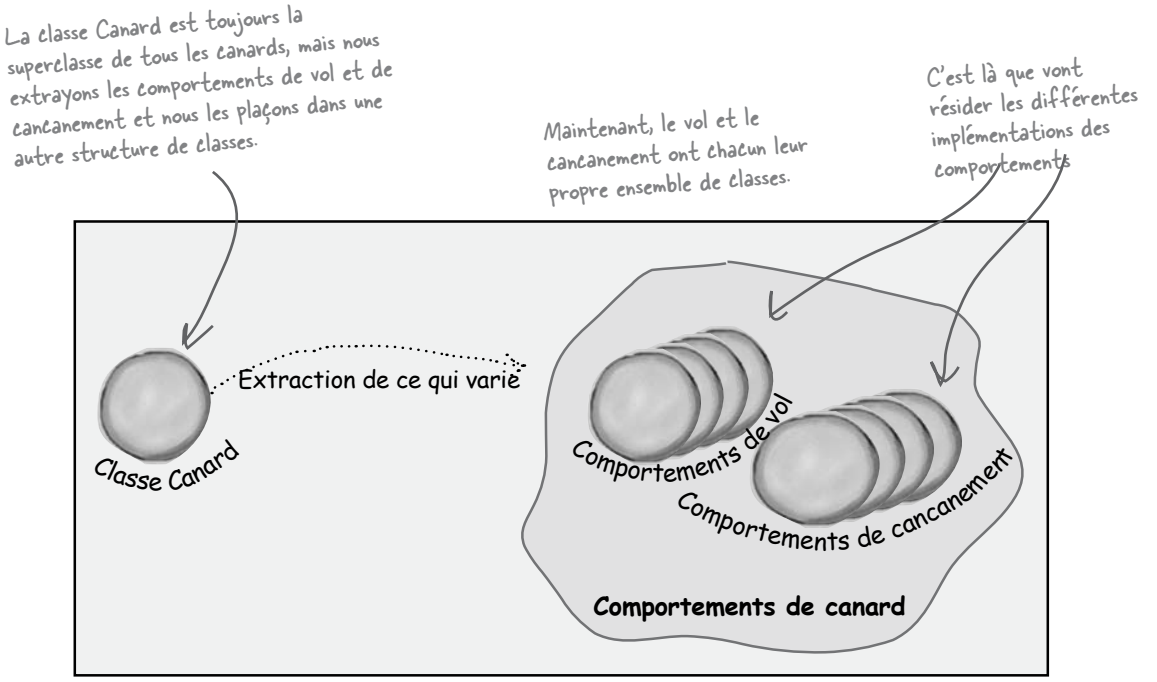
Séparer ce qui change de ce qui reste identique

Par où commencer ? Pour l'instant, en dehors des problèmes de voler() et de cancaner(), la classe Canard fonctionne bien et ne contient rien qui semble devoir varier ou changer fréquemment. À part quelques légères modifications, nous allons donc 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 distinctes de Canard), l'un pour *voler* et l'autre pour *cancaner*. Chaque ensemble de classes contiendra toutes les implémentations de leur comportement respectif. Par exemple, nous aurons une classe qui implémente le *cancanement*, *une autre* le *couinement* et *une autre* le *silence*.

Nous savons que voler() et cancaner() sont les parties de la classe Canard qui varient d'un canard à l'autre.

Pour séparer ces comportements de la classe Canard, nous extrayons ces deux méthodes de la classe et nous créons un nouvel ensemble de classes pour représenter chaque comportement.



Conception des comportements de canard

Comment allons-nous procéder pour concevoir les classes qui implémentent les deux types de comportements ?

Nous voulons conserver une certaine souplesse. Après tout, c'est d'abord la rigidité du comportement des canards qui nous a causé des ennuis. Et nous savons que nous voulons *affecter* des comportements aux instances des classes Canard. Par exemple instancier un nouvel objet Colvert et l'initialiser avec un *type* spécifique de comportement de vol. Et, pendant que nous y sommes, pourquoi ne pas faire en sorte de pouvoir modifier le comportement d'un canard dynamiquement ? Autrement dit, inclure des méthodes *set* dans les classes Canard pour pouvoir *modifier* la façon de voler du Colvert *au moment de l'exécution*.

Avec ces objectifs en tête, voyons notre deuxième principe de conception :

Principe de conception

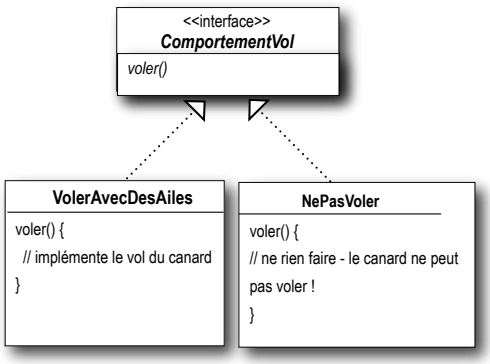
Programmer une interface, non une implémentation

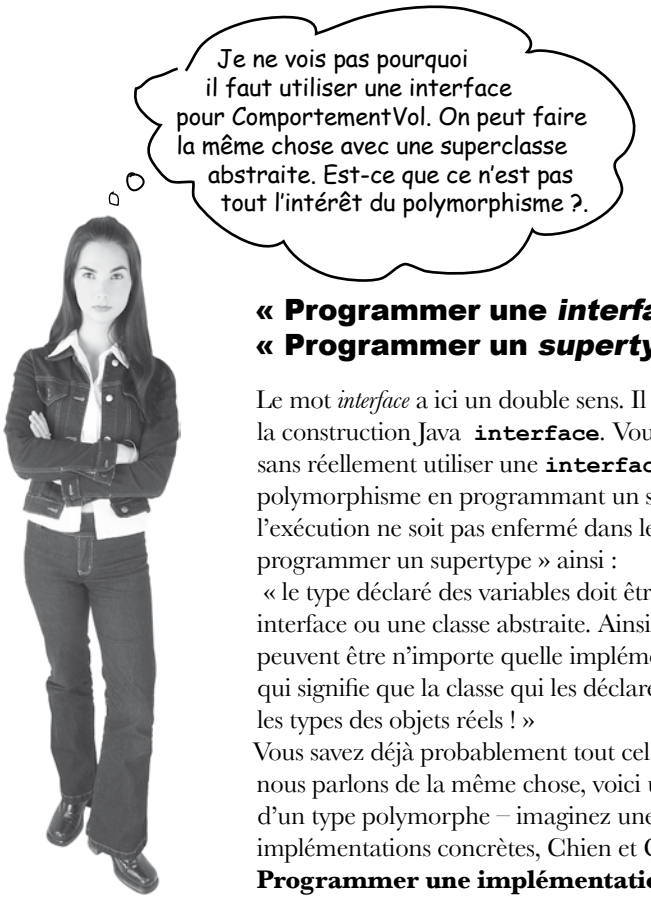
Nous allons utiliser une interface pour représenter chaque comportement – par exemple ComportementVol et ComportementCancan – et chaque implémentation d'un *comportement* implémentera l'une de ces interfaces. Cette fois, ce ne sont pas les classes *Canard* qui implémenteront les interfaces pour voler et cancaner. En lieu et place, nous allons créer un ensemble de classes dont la seule raison d'être est de représenter un comportement (par exemple « couiner »), et c'est la classe *comportementale*, et non la classe Canard, qui implémentera l'interface comportementale.

Ceci contraste avec notre façon ultérieure de procéder, dans laquelle un comportement provenait d'une implémentation concrète dans la superclasse Canard ou d'une implémentation spécialisée dans la sous-classe elle-même. Dans les deux cas, nous nous reposions sur une *implémentation*. Nous étions contraints à utiliser cette implémentation spécifique et il n'y avait aucun moyen de modifier le comportement (à part écrire plus de code).

Avec notre nouvelle conception, les sous-classes de Canard auront un comportement représenté par une *interface* (ComportementVol et ComportementCancan), si bien que l'*implémentation* réelle du comportement (autrement dit le comportement spécifique concret codé dans la classe qui implémente ComportementVol ou ComportementCancan) ne sera pas enfermé dans la sous-classe de Canard.

Désormais, les comportements de Canard résideront dans une classe distincte – une classe qui implémente une interface comportementale particulière.
Ainsi, les classes Canard n'auront besoin de connaître aucun détail de l'implémentation de leur propre comportement.





« Programmer une *interface* » signifie en réalité « Programmer un *supertype* ».

Le mot *interface* a ici un double sens. Il y a le *concept* d'interface, mais aussi la construction Java **`interface`**. Vous pouvez *programmer une interface* sans réellement utiliser une **`interface`** Java. L'idée est d'exploiter le polymorphisme en programmant un supertype pour que l'objet réel à l'exécution ne soit pas enfermé dans le code. Et nous pouvons reformuler « programmer un supertype » ainsi :

« 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 ! »

Vous savez déjà probablement tout cela, mais, juste pour être sûrs que nous parlons de la même chose, voici un exemple simple d'utilisation d'un type polymorphe – imaginez une classe abstraite Animal, avec deux implémentations concrètes, Chien et Chat.

Programmer une implémentation donnerait le code suivant :

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

Déclarer la variable "c" de type Chien (une implémentation concrète d'Animal) nous oblige à coder une implémentation concrète.

Mais **programmer une interface ou un supertype** donnerait ceci :

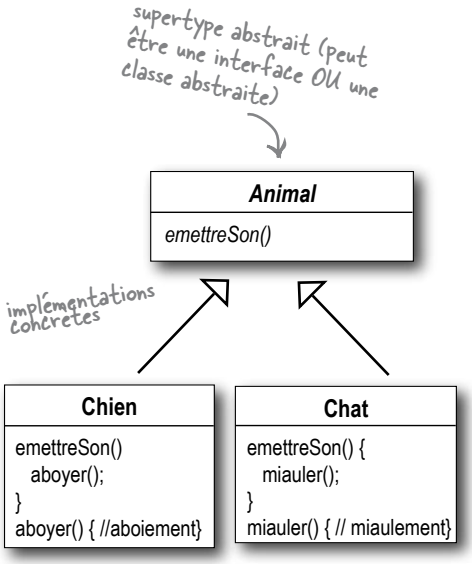
```
Animal animal = new Chien ();
animal.emettreSon();
```

Nous savons que c'est un Chien, mais nous pouvons maintenant utiliser la référence animal de manière polymorphe.

Mieux encore, au lieu de coder en dur l'instanciation du sous-type (comme new Chien ()) dans le code, **affectez l'objet de l'implémentation concrète au moment de l'exécution :**

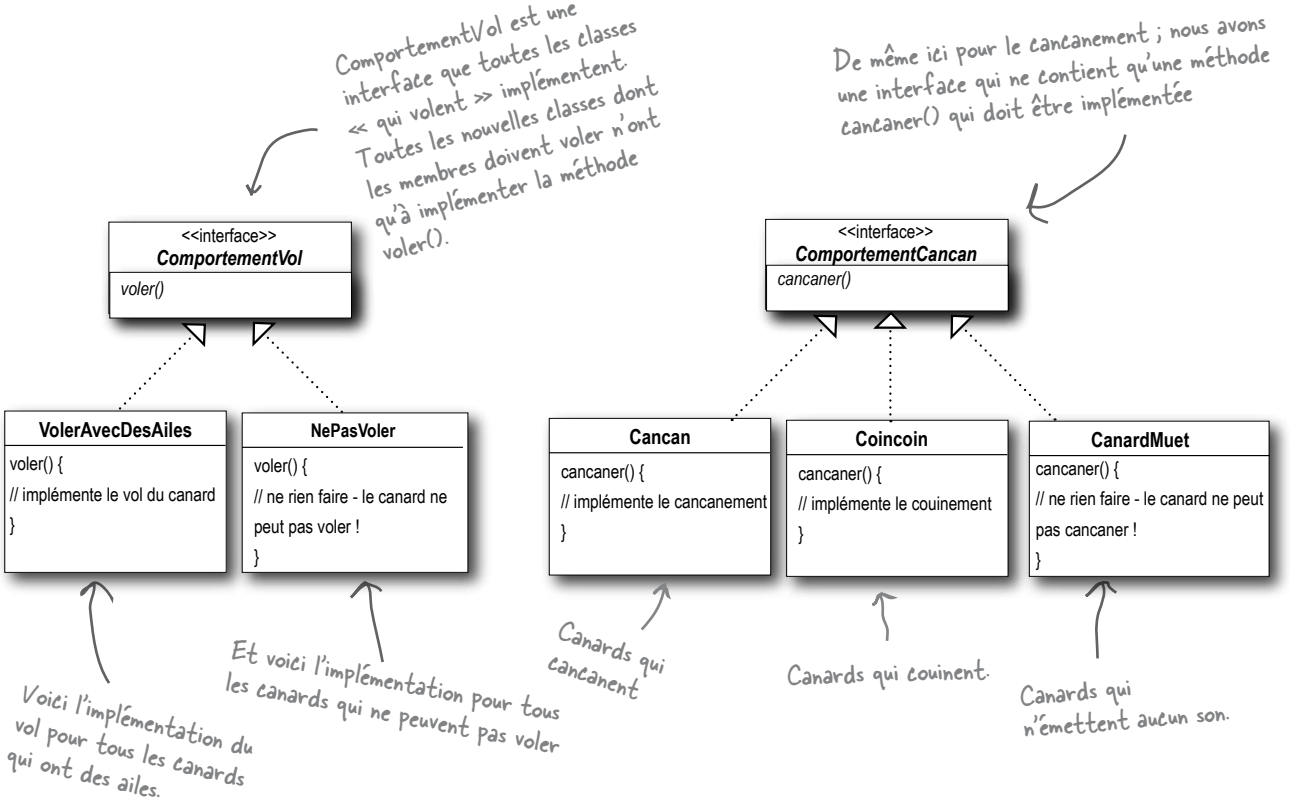
```
a = getAnimal();
a.emettreSon();
```

Nous ne savons pas QUEL EST le sous-type réel de l'animal... tout ce qui nous intéresse, c'est qu'il sait comment répondre à emettreSon().



Implémenter les comportements des canards

Nous avons ici deux interfaces, ComportementVol et ComportementCancan, ainsi que les classes correspondantes qui implémentent chaque comportement concret :



Avec cette conception, les autres types d'objets peuvent réutiliser nos comportements de vol et de cancanement parce que ces comportements ne sont plus cachés dans nos classes Canard !

Et nous pouvons ajouter de nouveaux comportements sans modifier aucune des classes comportementales existantes ni toucher à aucune des classes Canard qui utilisent les comportements de vol.

Nous obtenons ainsi les avantages de la RÉUTILISATION sans la surcharge qui accompagne l'héritage

Il n'y a pas de Questions Stupides

Q: 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 ?

R: Pas toujours. Quand vous concevez une application, vous anticipez souvent les points de variation, puis vous avancez et vous construisez le code de manière suffisamment souple pour pouvoir les gérer. Vous verrez que vous pouvez appliquer des principes et des patterns à n'importe quel stade du cycle de vie du développement.

Q: Est-ce qu'on devrait aussi transformer Canard en interface ?

R: Pas dans ce cas. Vous verrez qu'une fois que nous aurons tout assemblé, nous serons contents de disposer d'une classe concrète Canard et de canards spécifiques, comme Colvert, qui héritent des propriétés et des méthodes communes. Maintenant que nous avons enlevé de Canard tout ce qui varie, nous avons tous les avantages de cette structure sans ses inconvénients.

Q: 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 ?*

R: Dans un système OO, oui, les classes représentent des entités qui ont généralement un état (des variables d'instance) et des méthodes. Dans ce cas, *l'entité* se trouve être un comportement. Mais même un comportement peut avoir un état et des méthodes ; un comportement de vol peut avoir des variables d'instance qui représentent les attributs du vol (nombre de battements d'ailes par minute, altitude et vitesse maximales, etc.).

À vos crayons

1. En utilisant notre nouvelle conception, que feriez-vous pour ajouter la propulsion à réaction à l'application SuperCanard ?

2. Voyez-vous une classe qui pourrait utiliser le comportement de Cancan et qui n'est pas un canard ?

Réponses :

1) Créer une classe VolARéaction qui implémente l'interface ComportementVol.

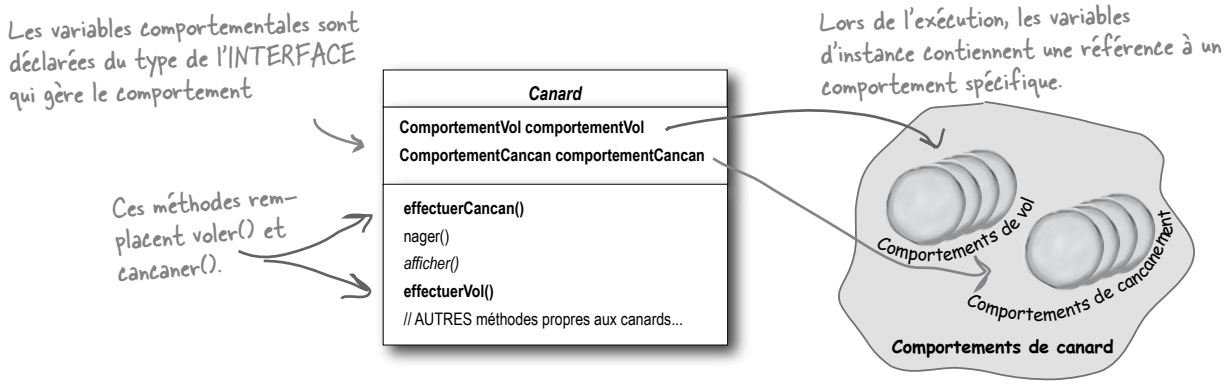
2) Par exemple un appeau (un instrument qui imite le cri du canard).

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).
Voici comment :

❶ **Nous allons d'abord ajouter à la classe Canard deux variables d'instance** nommées *comportementVol* et *comportementCancan*, qui sont déclarées du type de l'interface (non du type de l'implémentation concrète). Chaque objet Canard affectera ces variables de manière polymorphe pour référencer le type de comportement *spécifique* qu'il aimerait avoir à l'exécution (VolerAvecDesAiles, Coincoin, etc.).

Nous allons également ôter les méthodes voler() et cancaner() de la classe Canard (et de toutes ses sous-classes) puisque nous avons transféré ces comportements dans les classes ComportementVol et ComportementCancan. Nous allons remplacer voler() et cancaner() dans la classe Canard par deux méthodes similaires, nommées effectuerVol() et effectuerCancan(). Vous allez bientôt voir comment elles fonctionnent.



❷ **Implémentons maintenant effectuerCancan():**

```
public class Canard {
  ComportementCancan comportementCancan;
  // autres variables

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

Chaque Canard a une référence à quelque chose qui implémente l'interface ComportementCancan.

Au lieu de gérer son cancanement lui-même, l'objet Canard délègue ce comportement à l'objet référencé par comportementCancan. Les variables du comportement sont déclarées du type de l'INTERFACE comportementale.

Rien de plus simple, n'est-ce pas ? Pour cancaner, un Canard demande à l'objet référencé par comportementCancan de le faire à sa place.

Dans cette partie du code, peu nous importe de quelle sorte d'objet il s'agit.
Une seule chose nous intéresse : il sait cancaner !

Suite de l'intégration...

❸ Bien. Il est temps de s'occuper de la façon dont l**es variables d'instance comportementVol et comportementCancan sont affectées.** Jetons un coup d'œil à la classe Colvert :

```
public class Colvert extends Canard {

    public Colvert() {
        comportementCancan = new Cancan();
        comportementVol = new VolerAvecDesAiles();
     }

    }
```

Puisqu'un Colvert utilise la classe Cancan pour cancaner, quand effectuerCancan() est appelée, la responsabilité du cancanement est déléguée à l'objet Cancan et nous obtenons un vrai cancan.

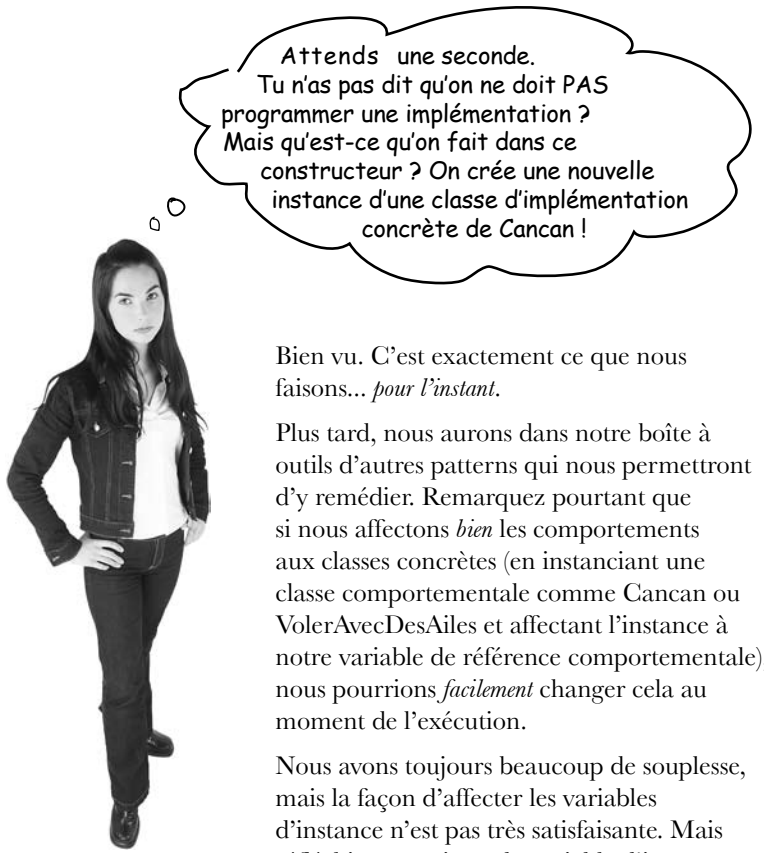
Et il utilise VolerAvecDesAiles comme type de ComportementVol.

Souvenez-vous que Colvert hérite les variables d'instance comportementCancan et comportementVol de la classe Canard.

```
    public void afficher() {
         System.out.println("Je suis un vrai colvert");
    }
}
```

Le cancan de Colvert est un vrai **cancan** de canard vivant, pas un **coincoin** ni un **cancan muet**. Quel est le mécanisme ? 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 ComportementCancan).

Et il en va de même pour le comportement de vol – le constructeur de Colvert ou de Mandarin initialise la variable d'instance comportementVol avec une instance de type VolerAvecDesAiles (une classe d'implémentation concrète de ComportementVol).



Bien vu. C'est exactement ce que nous faisons... *pour l'instant*.

Plus tard, nous aurons dans notre boîte à outils d'autres patterns qui nous permettront d'y remédier. Remarquez pourtant que si nous affectons *bien* les comportements aux classes concrètes (en instanciant une classe comportementale comme Cancan ou VolerAvecDesAiles et affectant l'instance à notre variable de référence comportementale), nous pourrions *facilement* changer cela au moment de l'exécution.

Nous avons toujours beaucoup de souplesse, mais la façon d'affecter les variables d'instance n'est pas très satisfaisante. Mais réfléchissez : puisque la variable d'instance comportementCancan est du type de l'interface, nous pourrons (par la magie du polymorphisme) affecter dynamiquement une instance d'une classe d'implémentation ComportementCancan différente au moment de l'exécution.

Faites une pause et demandez-vous comment vous implémenteriez un canard pour que son comportement change à l'exécution. (Vous verrez le code qui le permet dans quelques pages.)

Tester le code de Canard

1 Tapez et compilez le code de la classe Canard ci-après (Canard.java) et celui de la classe Colvert de la page 16 (Colvert.java).

```
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.

2 Tapez et compilez le code de l'interface ComportementVol (ComportementVol.java) et les deux classes d'implémentation comportementales (VolerAvecDesAiles.java et NePasVoler.java).

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

L'interface que toutes les classes comportementales « qui volent » implémentent.

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

Implémentation du comportement de vol pour les canards qui VOLENT...

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

Implémentation du comportement de vol pour les canards qui ne VOLENT PAS (comme les canards en plastique et les leurres).

Tester le code de Canard (suite)...

❸ Tapez et compilez le code de l'interface ComportementCancan (ComportementCancan.java) et celui des trois classes d'implémentation comportementales (Cancan.java, CancanMuet.java et CoinCoin.java).

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

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

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

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

❹ tapez et compilez le code de la classe de test (MiniSimulateur.java).

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

Cette ligne appelle la méthode héritée effectuerCancan() de Colvert, qui délègue alors à ComportementCancan de l'objet (autrement dit appelle cancaner() sur la référence héritée comportementCancan du canard).

Puis nous faisons de même avec la méthode héritée effectuerVol() de Colvert.

❺ Exécutez le code !

```
Fichier Édition Fenêtre Aide Yadayadayada
%java MiniSimulateur
Cancan
Je vole !!
```

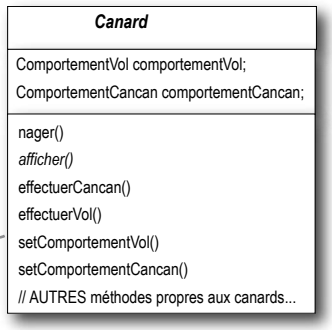
Modifier le comportement dynamiquement

Quel dommage que nos canards possèdent tout ce potentiel de dynamisme et qu'ils ne l'utilisent pas ! Imaginez que vous vouliez fixer le type de comportement du canard *via* des méthodes set dans la sous-classe de Canard au lieu de l'initialiser dans le constructeur.

❶ Ajoutez deux nouvelles méthodes à la classe Canard :

```
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.

note de l'éditeur : jeu de mots gratuit

❷ Créez un nouveau type de Canard (PrototypeCanard.java).

```
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.

❸ Créez un nouveau type de ComportementVol (PropulsionAReaction.java).

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 !");
    }
}
```

4 **Modifiez la classe de test (MiniSimulateur.java), ajoutez PrototypeCanard et munissez-le d'un réacteur.**

```
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();
    }
}
```

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

avant

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

après

5 **Exécutez-le !**

```
Fichier Édition Fenêtre Aide Yabadabadoo
%java MiniSimulateur
Cancan
Je vole !!
Je ne sais pas voler
Je vole avec un réacteur !
```

Pour modifier le comportement d'un canard au moment de l'exécution, il suffit d'appeler la méthode set correspondant à ce comportement.

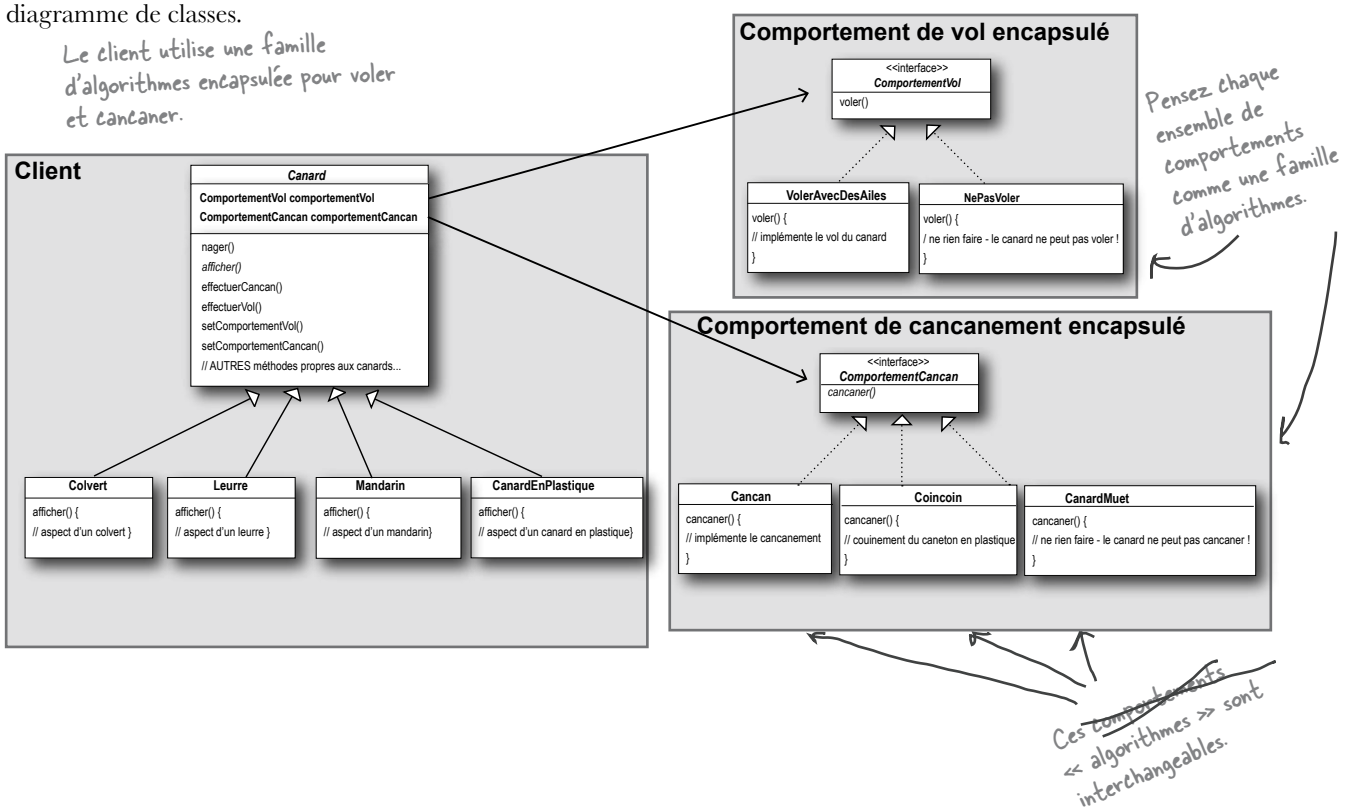
Vue d'ensemble des comportements encapsulés

Bien. Maintenant que nous avons plongé au cœur de la conception du simulateur, il est temps de venir respirer à la surface et de jeter un coup d'œil global.

Vous trouverez ci-après toute la structure de classes retravaillée. Nous avons tout ce qu'il nous faut : des canards qui dérivent de Canard, des comportements de vol qui implémentent ComportementVol et des comportements de cancanement qui implémentent ComportementCancan.

Remarquez aussi que nous avons commencé à décrire les choses un peu différemment. Au lieu de penser les comportements des canards comme un *ensemble de comportements*, nous allons commencer à les penser comme une *famille d'algorithmes*. Réfléchissez-y : dans la conception de SuperCanard, les algorithmes représentent ce qu'un canard ferait différemment (différentes façons de voler ou de cancaner), mais nous pourrions tout aussi bien utiliser les mêmes techniques pour un ensemble de classes qui implémenteraient les différentes façons de calculer la taxe d'habitation selon les différentes communes.

Soyez très attentif aux *relations* entre les classes. Prenez un crayon et notez la relation appropriée (EST-UN, A-UN et IMPLÉMENTE) sur chaque flèche du diagramme de classes.



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 ils 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.

Cette technique est importante ; en fait, nous avons appliqué notre troisième principe de conception :

Principe de conception

Préférez la composition à l'héritage

Comme vous l'avez constaté, créer des systèmes en utilisant la composition procure beaucoup plus de souplesse. Non seulement cela vous permet d'encapsuler une famille d'algorithmes dans leur propre ensemble de classes, mais vous pouvez également ***modifier le comportement au moment de l'exécution*** tant que l'objet avec lequel vous composez implémente la bonne interface comportementale.

La composition est utilisée dans de nombreux design patterns et vous en apprendrez beaucoup plus sur ses avantages et ses inconvénients tout au long de cet ouvrage.

MUSCLEZ VOS NEURONES

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 ?

Maître et disciple...

Maître : *Petit scarabée, dis-moi ce que tu as appris sur la Voie de l'orientation objet.*

Disciple : *Maître, j'ai appris que la promesse de la Voie de l'orientation objet était la réutilisation.*

Maître : *Continue, scarabée...*

Disciple : *Maître, l'héritage permet de réutiliser toutes les bonnes choses, et nous parviendrons à réduire radicalement les temps de développement et à programmer aussi vite que nous coupons le bambou dans la forêt.*

Disciple : *Scarabée, consacre-t-on plus de temps au code **avant** que le développement ne soit terminé ou **après** ?*

Disciple : *La réponse est **après**, maître. Nous consacrons toujours plus de temps à la maintenance et à la modification des logiciels qu'à leur développement initial.*

Maître : *Alors, scarabée, doit on placer la réutilisation **au-dessus** de la maintenabilité et de l'extensibilité ?*

Disciple : *Maître, je commence à entrevoir la vérité.*

Maître : *Je vois que tu as encore beaucoup à apprendre. Je veux que tu ailles méditer un peu plus sur l'héritage. Comme tu l'as constaté, l'héritage a des inconvénients et il y a d'autres moyens de parvenir à la réutilisation.*

À propos des design patterns...

Félicitations pour votre premier pattern !

Vous venez d'appliquer votre premier design pattern, le pattern STRATEGIE. Oui, vous avez utilisé le pattern Stratégie pour revoir la conception de l'application SuperCanard. Grâce à ce pattern, le simulateur est prêt à recevoir toutes les modifications que les huiles pourraient concocter lors de leur prochain séminaire aux Baléares.

Nous n'avons pas pris le plus court chemin pour l'appliquer, mais en voici la définition formelle :

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.

Ressortez CETTE définition quand vous voudrez impressionner vos amis ou influencer votre patron.

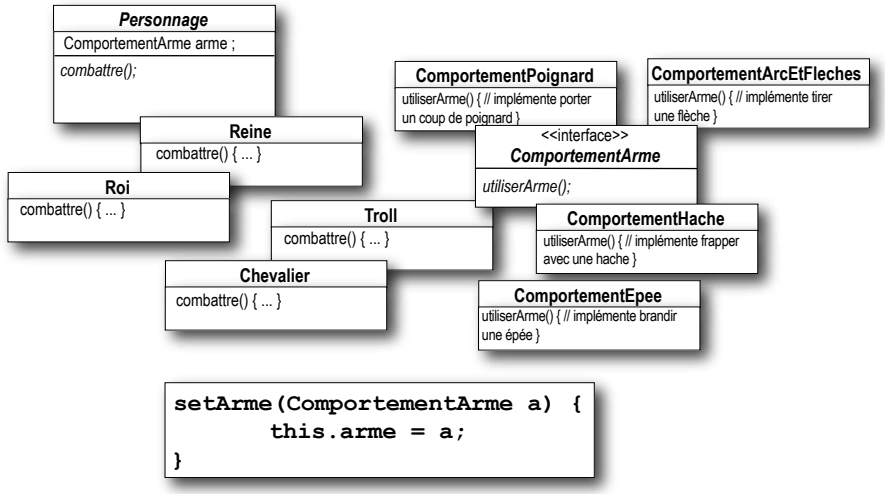
Problème de conception

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 pour les comportements correspondant aux armes que les personnages peuvent utiliser. Chaque personnage ne peut faire usage que d'une arme à la fois, mais il peut en changer à tout moment en cours de jeu. Votre tâche consiste à les trier...

(Les réponses se trouvent à la fin du chapitre.)

Votre tâche :

1. Réorganiser les classes.
2. Identifier une classe abstraite, une interface et huit classes ordinaires.
3. Tracer des flèches entre les classes.
   a. Tracez ce type de flèche pour l'héritage (« extends »).
   b. Tracez ce type de flèche pour l'interface (« implements »).
   c. Tracez ce type de flèche pour «A-UN».
4. Placer la méthode setArme() dans la bonne classe.



Entendu à la cafétéria...

Quelle est la différence entre ces deux commandes ? Aucune ! Elles sont parfaitement identiques, sauf qu'Alice utilise quatre fois plus de mots et risque d'épuiser la patience du ronchon qui prend les commandes.

Que possède donc Flo qu'Alice n'a pas ? **Un vocabulaire commun** avec le serveur. Non seulement il leur est plus facile de communiquer, mais le serveur a moins à mémoriser, puisqu'il a tous les « patterns » de la crêperie en tête.

Les design patterns vous permettent de partager un vocabulaire avec les autres développeurs. Une fois que vous disposez de ce vocabulaire, vous communiquez plus facilement avec eux et vous donnez envie de découvrir les patterns à ceux qui ne les connaissent. Cela vous permet également de mieux appréhender les problèmes d'architecture en **pensant au niveau des *patterns*** (des structures) et pas au niveau des détails, des *objets*.

Entendu dans le box voisin...

Alors, j'ai créé cette classe Diffusion. Elle conserve la trace de tous les objets qui l'écoutent, et, chaque fois qu'une nouvelle donnée arrive, elle envoie un message à chaque auditeur. Ce qui est génial, c'est que les auditeurs peuvent se joindre à la diffusion à toutmoment, et qu'ils peuvent même se désabonner. C'est une conception vraiment dynamique et faiblement couplée !

MUSCLEZ VOS NEURONES

Connaissez-vous d'autres vocabulaires communs en dehors de la conception OO et des cafétérias ? (Pensez aux garagistes, aux charpentiers, aux grands chefs, au contrôle de trafic aérien). Qu'est-ce que le jargon permet de communiquer ?

Quels sont les aspects de la conception OO qui sont véhiculés par les noms des patterns ? Quelles sont les qualités évoquées par le nom « Pattern Stratégie » ?

Paul, pourquoi ne pas dire simplement que tu appliques le pattern **Observateur** ?

Exactement. Si vous communiquez avec des patterns, les autres développeurs savent immédiatement et précisément de quoi vous parlez. Mais prenez garde à la « patternite »... Vous savez que vous l'avez contractée quand vous commencez à appliquer des patterns pour programmer Hello World...

Le pouvoir d'un vocabulaire commun

Lorsque vous communiquez en utilisant des patterns, vous faites plus que partager un JARGON.

Les vocabulaires partagés sont PUISSANTS.

Quand vous communiquez avec un autre développeur ou avec votre équipe en employant un nom de pattern, vous ne communiquez pas seulement un mot mais tout un ensemble de qualités, de caractéristiques et de contraintes que le pattern représente.

« Nous appliquons le pattern Stratégie pour implémenter les différents comportements de nos canards. » Cet énoncé vous informe que les comportements de canard ont été encapsulés dans un groupe de classes distinct, facile à étendre et à modifier, même, si nécessaire, au moment de l'exécution.

Les patterns vous permettent d'exprimer plus de choses en moins de mots. Quand vous utilisez un pattern dans une description, les autres développeurs comprennent tout de suite avec précision la conception que vous avez en tête.

Parler en termes de patterns permet de rester plus longtemps « dans la conception ». Décrire un système logiciel en termes de patterns vous permet de demeurer au niveau de la conception plus longtemps, sans devoir plonger dans les petits détails de l'implémentation des objets et des classes.

Combien avez-vous vu de réunions de conception qui s'enlisaient rapidement dans des détails d'implémentation ?

Un vocabulaire commun peut turbopropulser votre équipe de développement. Une équipe versée dans les design patterns avance beaucoup plus rapidement grâce à l'élimination des malentendus.

À mesure que votre équipe commencera à partager ses idées et son expérience en termes de patterns, vous construirez une communauté de patterns.

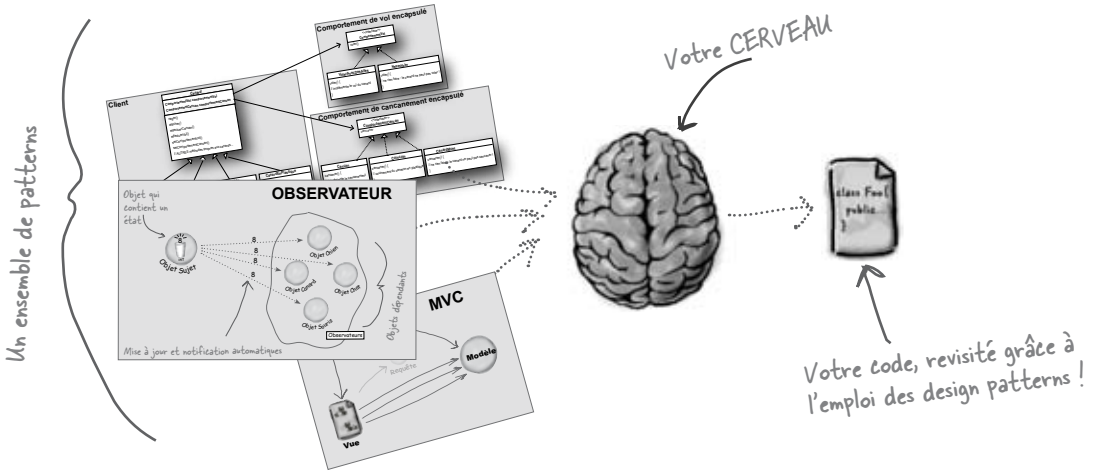
Un vocabulaire commun incite les développeurs juniors à apprendre plus vite. Les développeurs juniors écoutent les développeurs expérimentés. Quand les développeurs senior utilisent des design patterns, les juniors sont plus motivés pour les apprendre. Construisez une communauté d'utilisateurs de patterns dans votre société.

Pensez à lancer un groupe d'étude des patterns dans votre société. Peut-être même pourriez-vous être payé à apprendre...

Comment utiliser les design patterns ?

Nous avons tous utilisé des bibliothèques et des frameworks préexistants. Nous les prenons, écrivons le code en utilisant leur API, le compilons dans nos programmes et tirons parti d'une grande quantité de code que quelqu'un d'autre a écrit. Pensez au API Java et à toutes les fonctionnalités qu'elles vous offrent : réseau, interfaces utilisateurs, E/S, etc. Les bibliothèques et les frameworks font gagner beaucoup de temps dans la construction d'un modèle de développement où nous pouvons nous contenter de choisir les composants que nous insérerons directement. Mais... ils ne nous aident pas à structurer nos propres applications de façon qu'elles soient plus souples, plus faciles à comprendre et à maintenir. C'est là que les design patterns entrent en scène.

Les design patterns ne s'intègrent pas directement dans votre code, ils passent d'abord par votre CERVEAU. Une fois que vous avez chargé les patterns dans votre cerveau et que vous vous débrouillez bien avec, vous pouvez commencer à les appliquer à vos propres conceptions et à retravailler votre ancien code quand vous constatez qu'il commence à se rigidifier et à se transformer en un inextricable fouillis de code spaghetti.



Il n'y a pas de Questions Stupides

Q: 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 ?

R: Les design patterns sont au-dessus des bibliothèques. Ils nous indiquent comment structurer les classes et les objets pour résoudre certains problèmes. C'est à nous de les adapter à nos applications.

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

R: Les bibliothèques et les frameworks ne sont pas des design patterns : ils fournissent des implémentations spécifiques que nous lions à notre code. Mais il arrive que les bibliothèques et les frameworks s'appuient sur les design patterns dans leurs implémentations. C'est super, parce qu'une fois que vous aurez commencé à comprendre les patterns, vous maîtriserez plus vite les API qui sont structurées autour de patterns.

Q: Alors, il n'y a pas de bibliothèques de design patterns ?

R: Non, mais vous allez découvrir qu'il existe des catalogues qui énumèrent les patterns que vous pouvez employer dans vos applications.

Développeur sceptique

Gourou bienveillant

Développeur : O.K, hmm, pourquoi n'est-ce pas seulement une affaire de bonne conception objet ? Je veux dire, tant que j'applique l'encapsulation et que je connais l'abstraction, l'héritage et le polymorphisme, est-ce que j'ai vraiment besoin des design patterns ? Est-ce que ce n'est pas plus simple que cela ? Est-ce que ce n'est pas la raison pour laquelle j'ai suivi tous ces cours sur l'OO ? Je crois que les design patterns sont utiles pour ceux qui connaissent mal la conception OO.

Gourou : Ah, c'est encore un de ces malentendus du développement orienté objet : connaître les bases de l'OO nous rend automatiquement capables de construire des systèmes souples, réutilisables et faciles à maintenir.

Développeur : Non ?

Gourou : Non. En l'occurrence, la construction de systèmes OO possédant ces propriétés n'est pas toujours évidente, et seul beaucoup de travail a permis de la découvrir.

Développeur : Je crois que je commence à saisir. Ces façons de construire des systèmes orientés objets pas toujours évidentes ont été collectées...

Gourou : Oui, et constituent un ensemble de patterns nommés Design Patterns.

Développeur : Et si je connais les patterns, je peux me dispenser de tout ce travail et sauter directement à des conceptions qui fonctionnent tout le temps ?

Gourou : Oui, jusqu'à un certain point. Mais n'oublie pas que la conception est un art. Un pattern aura toujours des avantages et des inconvénients. Mais si tu appliques des patterns bien conçus et qui ont fait leurs preuves au fil du temps, tu auras toujours beaucoup d'avance.

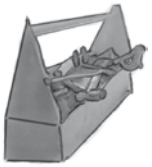
Souvenez-vous que la connaissance de concepts comme l'abstraction, l'héritage et le polymorphisme ne fait pas de vous un bon concepteur orienté objet. Un gourou de la conception réfléchit à la façon de créer des conceptions souples, faciles à maintenir et qui résistent au changement.

Développeur : Et si je ne trouve pas de pattern ?

Gourou : Les patterns sont sous-tendus par des principes orientés objet. Les connaître peut t'aider quand tu ne trouves pas de pattern qui corresponde à ton problème.

Développeur : Des principes ? Tu veux dire en dehors de l'abstraction, de l'encapsulation et...

Gourou : Oui, l'un des secrets de la création de systèmes OO faciles à maintenir consiste à réfléchir à la façon dont ils peuvent évoluer, et ces principes traitent de ces problèmes.



Votre boîte à outils de concepteur

Vous avez presque terminé le premier chapitre ! Vous avez déjà mis quelques outils dans votre boîte à outils OO. Récapitulons-les avant de passer au chapitre 2.

Bases de l'OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage

Nous supposons que vous connaissez les concepts OO de base : comment organiser les classes en exploitant le polymorphisme, comment l'héritage est une sorte de conception par contrat et comment l'encapsulation fonctionne. Si vous êtes un peu rouillé sur ces sujets, ressortez votre livre Java tête la première et revoyez-les, puis relisez rapidement ce chapitre.

Principes OO

- Encapsulez ce qui varie.
- Préférez la composition à l'héritage.
- Programmez des interfaces, non des implémentations.

Nous les reverrons plus en détail et nous en ajouterons d'autres à la liste.

PatternsOO

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.

Tout au long de ce livre, réfléchissez à la façon dont les patterns s'appuient sur les concepts de base et les principes OO.

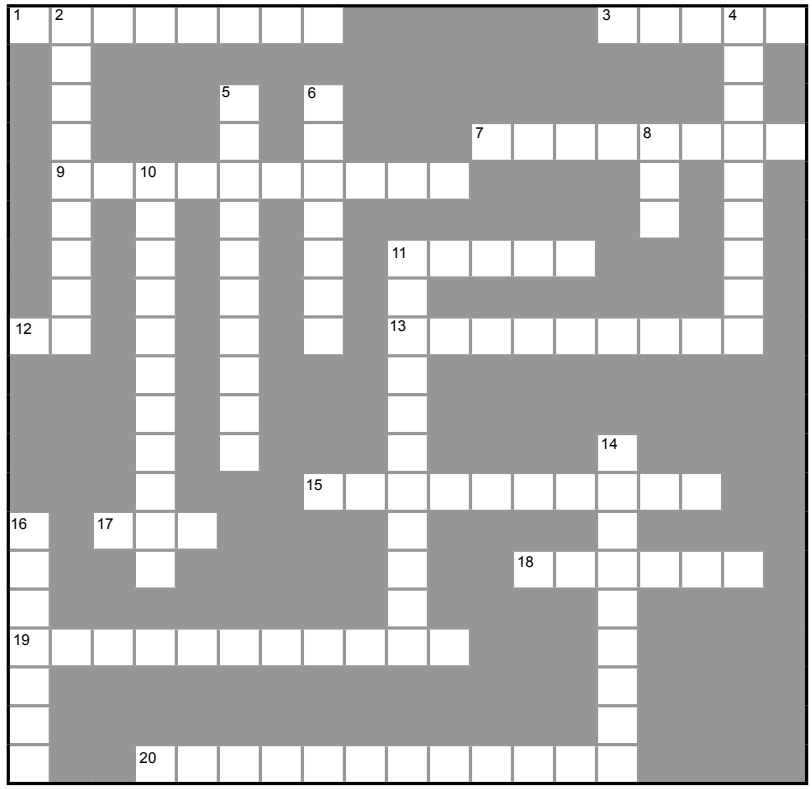
En voici un, d'autres suivront !

POINTS D'IMPACT

- Connaître les bases de l'OO ne fait pas de vous un bon concepteur.
- Les bonnes conceptions OO sont souples, extensibles et faciles à maintenir.
- Les patterns vous montrent comment construire des systèmes OO de bonne qualité.
- Les patterns résument une expérience éprouvée de la conception objet.
- Les patterns ne contiennent pas de code, mais des solutions génériques aux problèmes de conception. Vous devez les adapter aux applications spécifiques.
- Les patterns ne sont pas *inventés*, ils sont *découverts*.
- La plupart des patterns et des principes traitent des problèmes de changement dans le logiciel.
- La plupart des patterns permettent à une partie d'un système de varier indépendamment de toutes les autres.
- On essaie souvent d'extraire ce qui varie d'un système et de l'encapsuler.
- Les patterns fournissent un langage commun qui peut optimiser votre communication avec les autres développeurs.

Donnons à votre cerveau droit quelque chose à faire.

Ce sont vos mots-croisés standard. Tous les mots de la solution sont dans ce chapitre.



Horizontalement

1. Méthode de canard.
3. Modification abrégée.
7. Les actionnaires y tiennent leur réunion.
9. _____ ce qui varie.
11. Java est un langage orienté _____.
12. Dans la commande de Flo.
13. Pattern utilisé dans le simulateur.
15. Constante du développement.
17. Comportement de canard.
18. Maître.
19. Les patterns permettent d'avoir un _____ commun.
20. Les méthodes set permettent de modifier le _____.d'une classe.

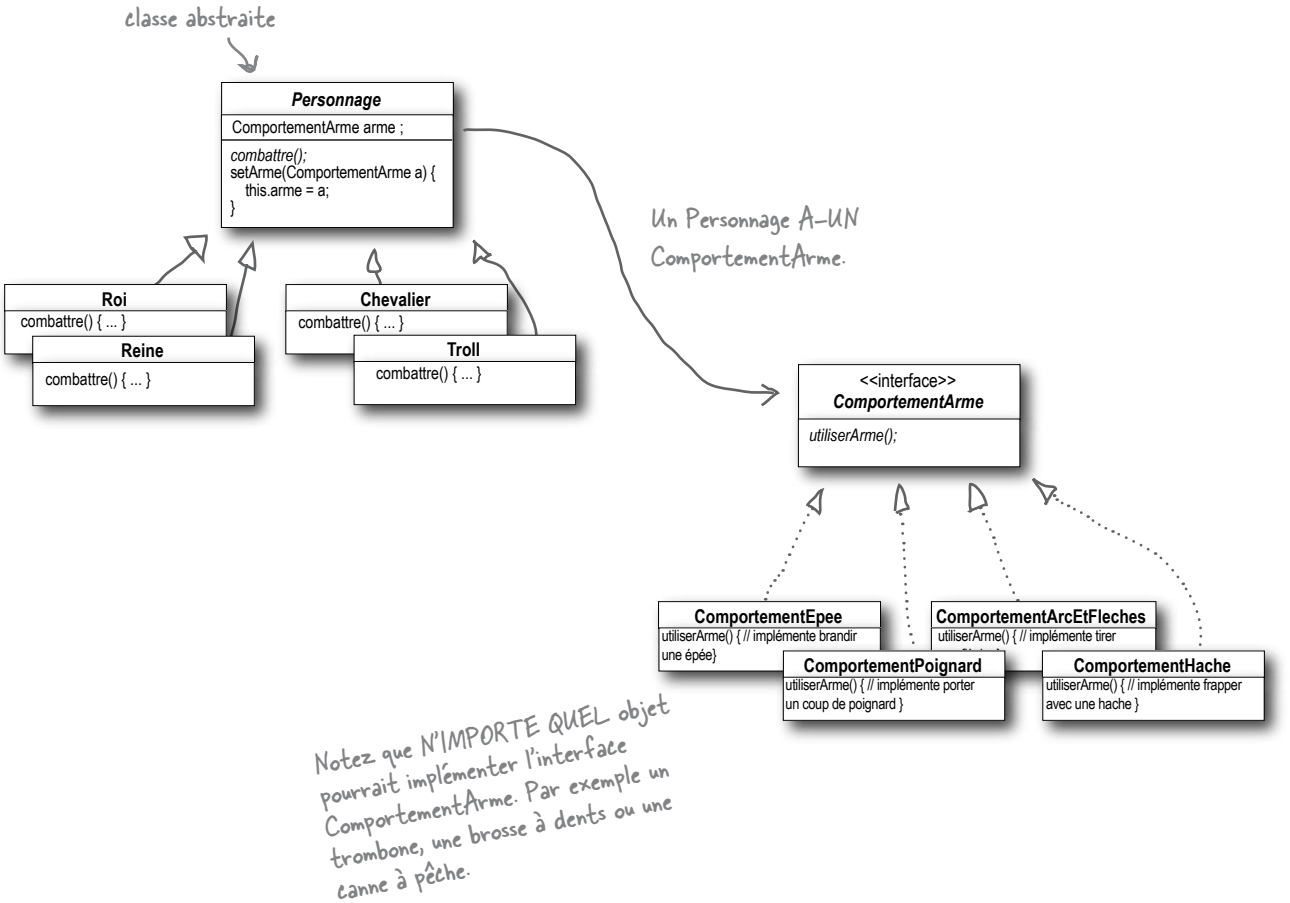
Verticalement

2. Bibliothèque de haut niveau.
4. Programmez une _____, non une implémentation.
5. Les patterns la synthétisent.
6. Ils ne volent ni ne cancanent.
8. Réseau, E/S, IHM.
10. Préférez-la à l'héritage.
11. Paul applique ce pattern.
14. Un pattern est une solution à un problème _____.
16. Sous-classe de Canard.

Solution du problème de conception

Personnage est la superclasse abstraite de tous les autres personnages (Roi, Reine, Chevalier et Troll) tandis que ComportementArme est une interface que toutes les armes implémentent. En conséquence, tous les personnages et toutes les armes sont des classes concrètes.

Pour changer d'arme, chaque personnage appelle la méthode setArme() qui est définie dans la superclasse Personnage. Lors d'un combat, la méthode utiliserArme() est appelée sur l'arme courante d'un personnage donné afin d'infliger de grands dommages corporels à un autre personnage.

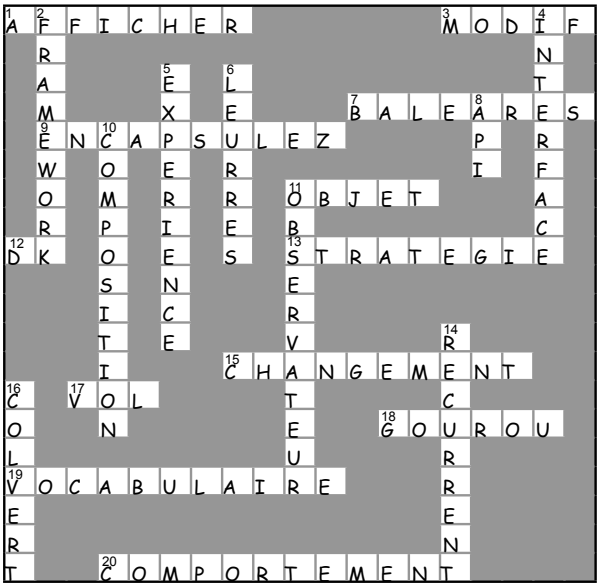


Solutions

À vos crayons

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

- [x] A. Le code est dupliqué entre les sous-classes.
- [x] B. Les changements de comportement au moment de l'exécution sont difficiles.
- [ ] C. Nous ne pouvons pas avoir de canards qui dansent.
- [x] C. Il est difficile de connaître tous les comportements des canards.
- [ ] D. Les canards ne peuvent pas voler et cancaner en même temps.
- [x] E. Les modifications peuvent affecter involontairement d'autres canards.



À vos crayons

Quels sont les facteurs qui peuvent induire des changements dans vos applications ? Votre liste est peut-être différente, mais voici quelques-unes de nos réponses. Est-ce que cela vous rappelle quelque chose ?

Mes clients ou mes utilisateurs décident qu'ils veulent autre chose ou qu'ils ont besoin d'une nouvelle fonctionnalité.

Mon entreprise a décidé qu'elle allait changer de système de SGBD et qu'elle allait également acheter ses données chez un autre fournisseur dont le format est différent. Argh !

La technologie évolue et nous devons mettre à jour notre code afin d'utiliser d'autres protocoles.

Après tout ce que nous avons appris en construisant notre système, nous aimerions revenir en arrière et intégrer quelques améliorations.