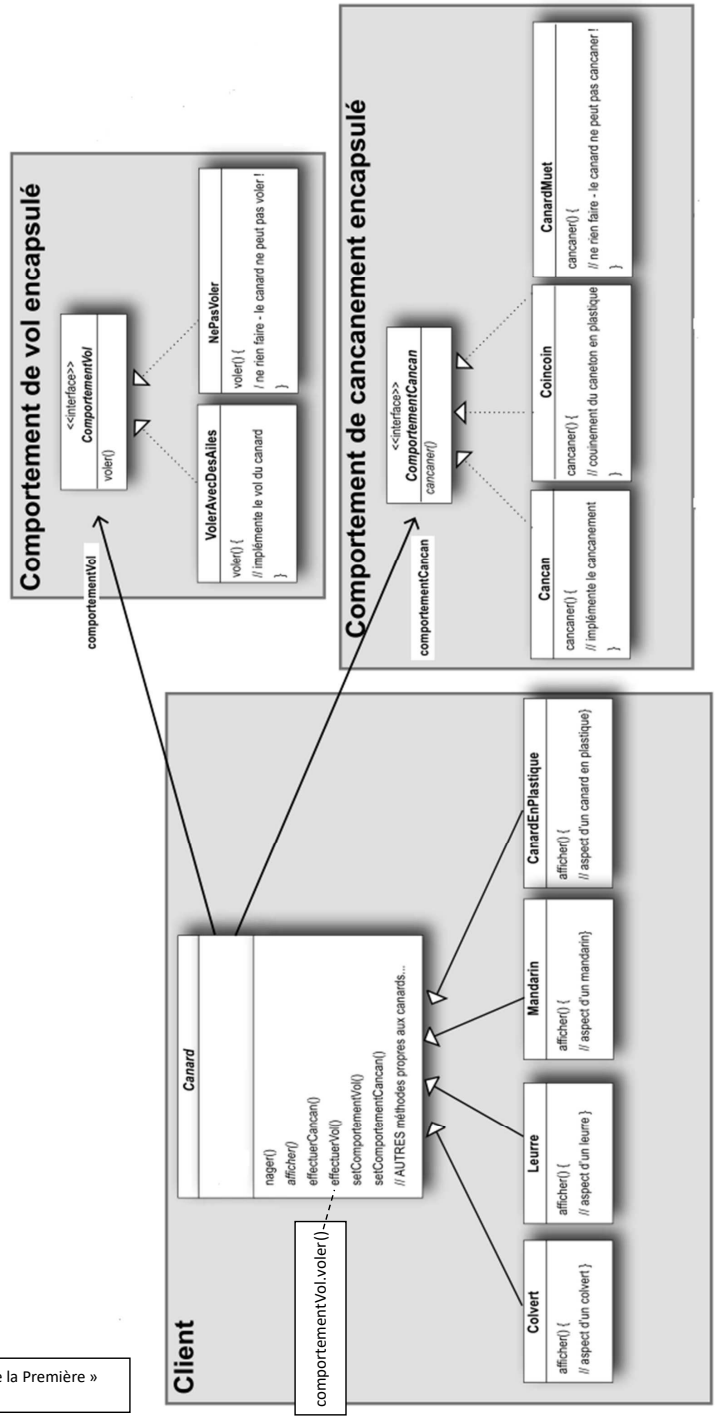
M2104 – TD : Diagrammes de classes, Génération de code & Tests (Partie n°2)

Et si on cancanait un peu mieux …

… Au TD précédent, il vous a été demandé de modéliser le système suivant …

Vous allez participer au développement d'un jeu de simulation de mare aux canards.
Le jeu doit pouvoir afficher toutes sortes de canards. Les canards peuvent `nager` et `cancaner` (c-a-d émettre des sons).
Chaque sous-type de canard (`Colvert`, `Mandarin`, `Canard en plastique`, `Leurre`, …) est chargé de s'`afficher` correctement, lui seul connait la manière dont il apparaît à l'écran (dans un premier temps l'affichage se limitera à une phrase du style "Je suis un vrai colvert" , "Je suis un vrai mandarin" , …)
Tous les canards peuvent `nager` (et oui, tous les canards flottent, même les leurres!)…
Les canards doivent aussi pouvoir `voler`… mais attention, pas tous : les canards en plastique ne doivent pas voler, cela ferait désordre de voir des canards en plastique voler dans tout l'écran 😉
Les canards n'émettent pas tous le même son : un *vrai* canard (comme le colvert ou le mandarin) émet un cancanement (*can-can*), alors qu'un canard en plastique émet un couienement (*coin-coin*)… ah oui, et un leurre n'émettra aucun son (ce sera un canard muet 😉) .
Et pour finir, saviez-vous que durant l'hiver, le canard perd ses plumes de vol (rémiges) et ne peut donc pas voler pour un certain temps ?
Du coup, ce serait bien de faire en sorte qu'un canard ait un comportement de vol qui lui est propre et qui pourra être modifier dynamiquement au cours du jeu c-a-d n'importe quand durant le cycle de vie de cet objet.

Notre super équipe de concepteurs « Head First Team » a proposé la modélisation suivante pour répondre élégamment à toutes les contraintes de ce problème…



Source : « Design Patterns Tête la Première » de E&E Freeman

1. Commencez par comprendre cette modélisation.
   Reprenez l'énoncé du problème, et expliquez, point par point, en quoi la conception proposée ici répond à toutes les contraintes énoncées.

2. Notez, sur le modèle précédent, la relation appropriée (**EST-UN** ou **A-UN** ou **IMPLÉMENTE**) sur chaque flèche du diagramme de classes.

3. Comparez ce diagramme annoté au diagramme de classes annoté de l'exercice 2 (sur la stratégie pour les recherches) du TD précédent.
   a. Voyez-vous une similitude entre ces deux diagrammes ?
   b. Par rapport à cette similitude, proposez, dans un nouveau diagramme de classes, une architecture de classes et/ou interfaces qui reprend de manière générique cette similitude. Pour modéliser cette architecture vous utiliserez 4 classes et/ou interfaces que vous appellerez **`Contexte`** (avec une méthode `contexte()`), **`Strategie`**, **`StrategieConcreteA`** et **`StrategieConcreteB`** (avec leur méthode `algorithme()`)
   c. Une fois le diagramme modélisé, répondez aux questions suivantes :
      → A propos de **`Stratégie`** :
         Que fait **`Strategie`** ?
         Qui utilise **`Strategie`** et pourquoi ?
      → A propos d'une **`StrategieConcrete`** :
         Que fait une **`StrategieConcrete`** ?
      → A propos de **`Contexte`** :
         De quel objet est composé **`Contexte`** ?
         Comment **`Contexte`** gère-t-il la référence à un tel objet?

Remarque pour ceux qui auraient envie d'en savoir plus …
Un tel diagramme de classes écrit de manière générique est appelé un *modèle de conception*. Un modèle de conception (***design pattern***) répond à une problématique donnée qui est ici de pouvoir ***définir une famille d'algorithmes, encapsuler chacun d'entre eux et les rendre interchangeables***. Le pattern que vous venez d'écrire n'est autre que le **pattern Stratégie** qui permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent.
Le diagramme de classes (sous sa forme générique) peut ensuite être transposé dans n'importe quel contexte qui a besoin de résoudre une problématique similaire (contexte de la stratégie de recherche pour l'exercice 2 ou contexte de la marre aux canards pour l'exercice 3). La notion de ***Design Pattern*** sera étudiée l'année prochaine dans le module M3105 de programmation et conception orientée objet avancée #teasing 😉

4. a. Ecrire maintenant, sur une feuille, le code java des classes/interfaces suivantes : **`ComportementVol`**, **`VolerAvecDesAiles`** et **`Canard`**.
   Vous pouvez mettre **`//TODO`** comme implémentation des méthodes sauf pour `effectuerVol` , `effectuerCancan` , `setComportementVol` et `setComportementCancan` où vous devez écrire l'implémentation attendue 😉

   b. Ecrire également la classe **`Colvert`** :
   sachant que lorsqu'il vient au monde un petit colvert est un *vrai* canard qui a la capacité de voler avec de ses propres ailes.
   Le cancan du colvert est un vrai cancan de canard vivant, pas un coincoin ni un cancan muet.

5. Le client de la mare aux canard souhaite désormais faire évoluer son jeu en proposant un nouveau type de canard (`PrototypeCanard`), ainsi qu'un nouveau comportement de vol (`PropulsionAReaction`)
   a. Modifier le diagramme de classes de la page 2 pour faire apparaître ces deux nouvelles classes sur ce modèle. Est-ce facile à modifier ?
      Afin de tester l'ajout de ces nouvelles classes (et de vérifier la non-régression sur les anciennes), votre ami testeur vous demande d'implémenter le ***scénario de test*** suivant :

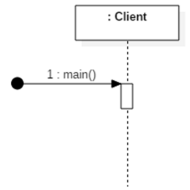
> ***Commencer par créer un colvert et demander au colvert d'effectuer un cancan, puis un vol***
>
> ***Puis, créer un prototype de canard et lui demander d'effecteur un vol***
>
> ***Modifier ensuite le comportement de vol ce prototype de canard qui doit pouvoir désormais voler avec une propulsion à réaction.***
>
> ***Demander au prototype de canard d'effectuer un vol.***

Remarque : On appelle ce genre de test un **test d'acceptance** ou un **test de recette** : il permet de s'assurer que le produit est conforme aux spécifications… N'hésitez pas à consulter la définition du test d'acceptance dans le cours sur les Tests 😉

Remarque : Le prototype de canard vient au monde sans aucun moyen de voler, ni aucun moyen de cancaner.

b.1 En considérant que les classes `PrototypeCanard` et `PropulsionAReaction` ont déjà été implémentées par un de vos collègues développeurs, écrire dans le `main` d'une classe `Client` le code java du scénario de test précédent.

b.2 Représentez sur un diagramme de séquences ***toutes*** les interactions (messages échangés entre les objets) lors de l'exécution de cette méthode `main`. Ce diagramme commencera de la manière suivante.
A vous de le compléter avec les objets et messages appropriés.



6. ***Un petit bilan :*** Les principes de bases de l'orienté objet sont : La **classe**, l'**encapsulation**, l'**héritage** et le **polymorphisme**.
   a. En vous aidant du cours d'introduction à la modélisation, donnez la définition de chacun de ces principes.
   b. Expliquez comment la conception précédente met en œuvre ses principes et en quoi une utilisation judicieuse de ces principes permet de faciliter l'extension du logiciel (comme l'ajout d'un nouveau comportement par exemple).