TP1

Rendu : une archive avec le code source en Java et un document au format PDF.

1 Héritage ou composition

Question 1. Ecrire quatre classes: `Mere`, `Fille`, `Pere` et `Fils`. La classe `Fille` hérite de `Mere`, tandis que la classe `Fils` possède une instance de la classe `Pere`. Autrement dit, on utilise l'*héritage* entre `Mere` et `Fille`, tandis qu'on utilise la *composition* entre `Pere` et `Fils`. Dessiner un diagramme de classe reflétant la situation.

Question 2. La classe `Mere` a une méthode `void compter()` qui affiche les nombres de 1 à 10. La classe `Fille` a un prénom et une méthode `String getPrenom()` renvoyant le prénom. Ecrire ces classes ainsi qu'une classe de test créant une instance de la classe `Fille` et produisant l'affichage suivant.

```
1 2 3 4 5 6 7 8 9 10
Lucie
```

L'affichage des nombres de 1 à 10 à été obtenu par héritage grâce à `super.compter()`.

Question 3. La classe `Pere` a un nom, un prénom et une méthode `String getNom()` renvoyant le nom. La classe `Fils` a un père, un prénom et une méthode `String getPrenom()`. De plus, elle a une méthode `String getNom()` qui délègue sa réponse à une instance de la classe `Pere` et un modificateur `void setPere(Pere pere)`. Ecrire ces classes et compléter la classe de test de sorte à ce qu'elle crée deux instances de la classe `Pere` et une de la classe `Fils` qui produise l'affichage suivant.

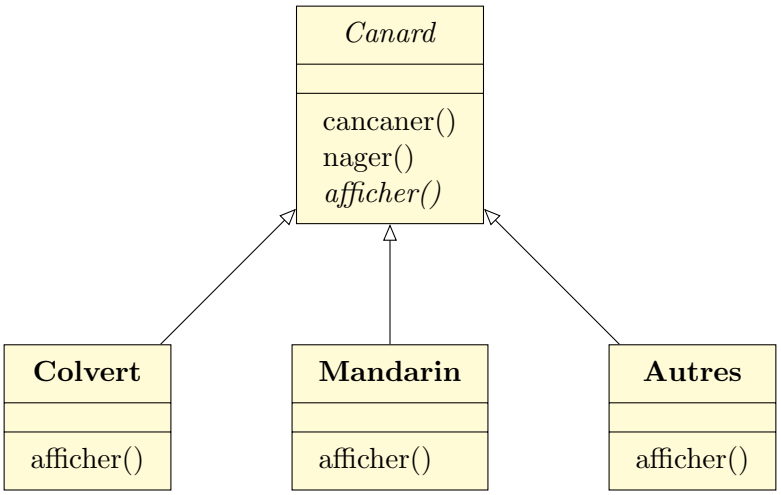
```
1 2 3 4 5 6 7 8 9 10
Lucie
=====
Adrien
Dupont
Durand
```

L'affichage des noms de famille a été obtenu par composition. Le comportement de la méthode `String getNom()` de la classe `Fils` a été changé de manière dynamique.

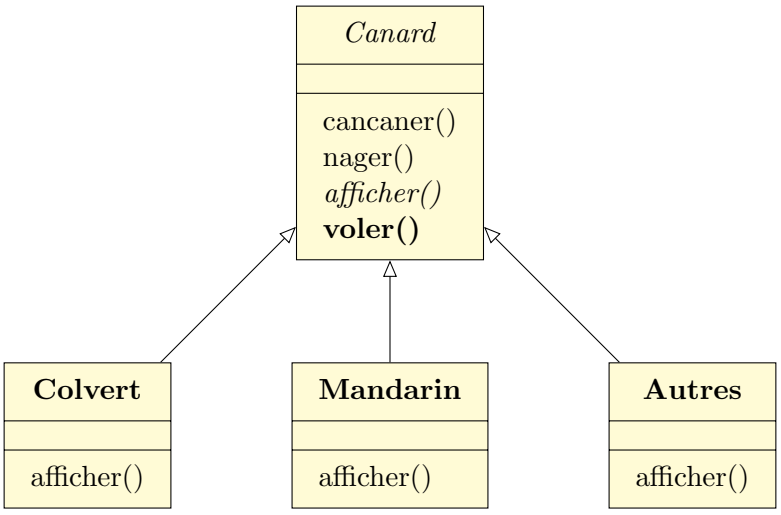
Question 4. Compléter le diagramme de classe avec les nouvelles informations.

2 Le design pattern Stratégie

Question 1. *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 orientées objet standard et créé une superclasse* `Canard` *dont tous les autres types de canards héritent.* Ecrire les classes `Canard`, `Colvert`, `Mandarin` et une classe de test dans laquelle une instance de `Colvert` et une de `Mandarin` appellent chacune leur méthode `afficher()`. Cette dernière indique le type du canard sur la sortie standard.



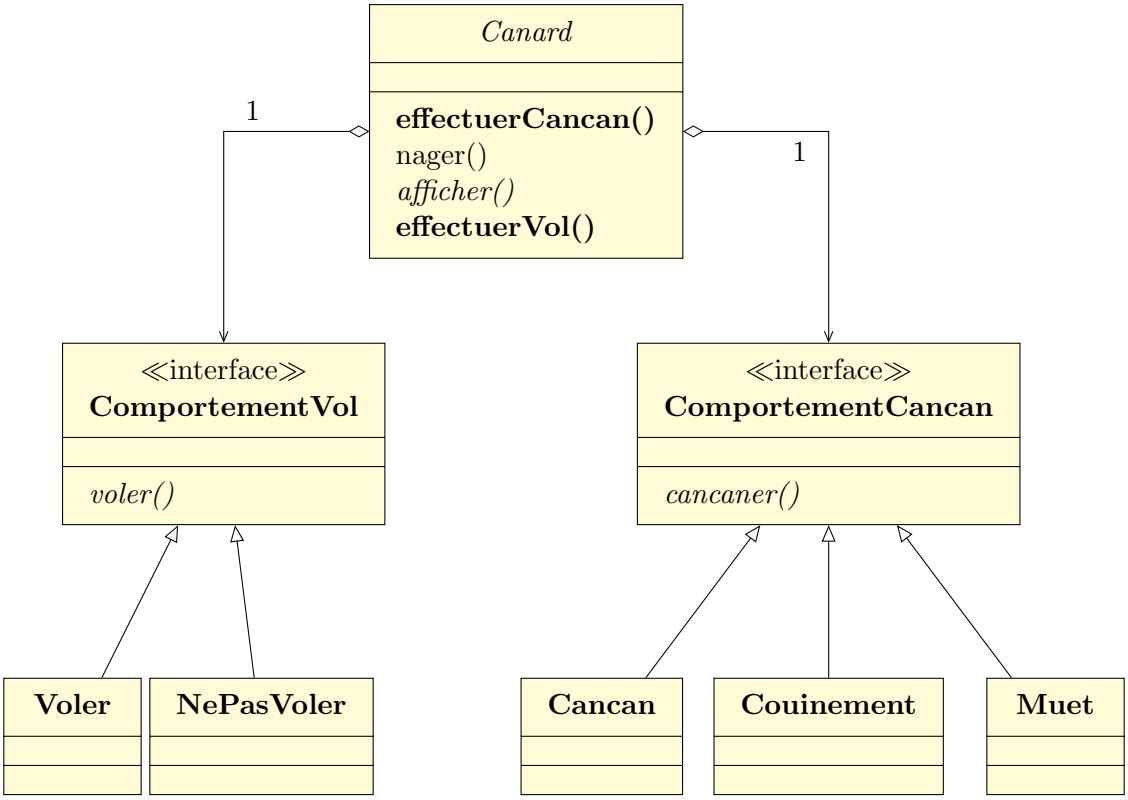
Question 2. *L'an passé, la société a subi de plus en plus de pression de la part de la concurrence. Les dirigeants ont décidé que des canards volants étaient exactement ce qu'il fallait pour battre la concurrence à plate couture. Naturellement, le responsable de Joël leur a dit que celui-ci n'aurait aucun problème pour bricoler quelque chose en une semaine. Joël ajoute une méthode* `voler()` *dans la classe* `Canard` *et tous les autres canards en hériteront.*



Joël a oublié que toutes les sous-classes de ***Canard*** *ne doivent pas voler, notamment les canards en plastique. Il y a un léger défaut dans la conception. Il se dit qu'il pourrait redéfinir* ***voler()*** *et* ***cancaner()*** *pour les canards en plastique. Il se trouve que ce n'est pas la seule exception : il devra faire le même travail pour les leurres.* Parmi les choix suivants, quels sont les inconvénients à utiliser l'héritage pour définir le comportement de `Canard` ?

1. Le code est dupliqué entre les sous- classes.
2. Les changements de comportement au moment de l'exécution sont difficiles.
3. Nous ne pouvons pas avoir de canards qui dansent.
4. Il est difficile de connaître tous les comportements des canards.
5. Les canards ne peuvent pas voler et cancaner en même temps.
6. Les modifications peuvent affecter involontairement d'autres canards.

Question 3. *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.*
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 des interfaces `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. En effet, chaque classe qui implémente l'interface devra définir la méthode en question, résultant en de la duplication de code. De plus, tous les canards qui doivent voler ne volent peut-être pas de la même façon.
Nous allons utiliser la *composition* et 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, et c'est la classe comportementale, et non la classe Canard, qui implémentera l'interface comportementale.



Réprendre le code pour qu'il corresponde à cette nouvelle conception.

Question 4. Ajouter deux classes de canards : `Leurre` et `Prototype`. `Prototype` dispose de deux modificateurs `setComportementVol` et `setComportementCancan`.

Question 5. Le vol peut à présent se faire par propulsion. Ajouter un nouveau comportement pour le vol : `Propulsion`.

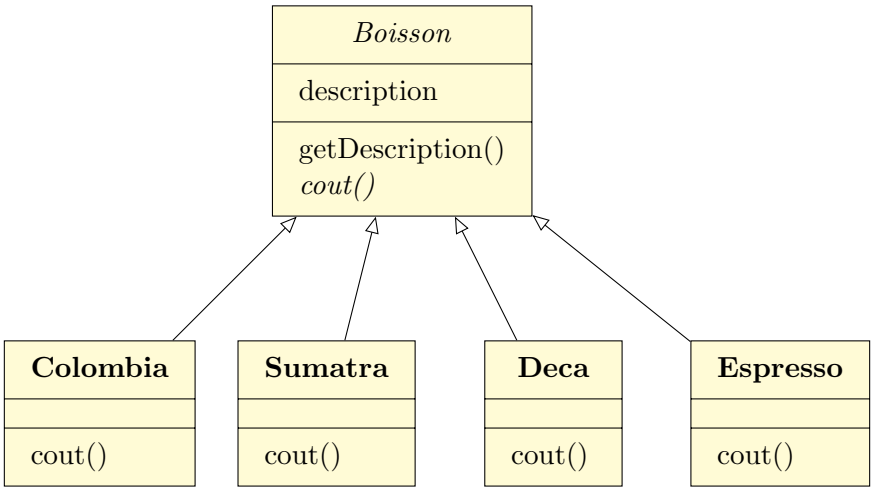
Question 6. Ecrire un test instanciant les classes `Colvert`, `Leurre` et `Prototype`. Pour chacun, appeler dans l'ordre les méthodes `afficher()`, `effectuerCancan()` et `effectuerVol()`. Modifer les comportements de l'instance de `Prototype`, puis appeler de nouveau les méthodes `effectuerCancan()` et `effectuerVol()`. Un exemple d'affichage est présent ci-dessous.

```
Je suis un colvert.
Cancan !
Je vole !
=====
Je suis un leurre.
...
Je ne vole pas.
=====
Je suis un prototype.
...
Je vole !
Coin coin !
Je vole par propulsion !
```

Question bonus. Rendre le code réellement dynamique en permettant à l'utilisateur de choisir les comportements.

3 Le design pattern Décorateur

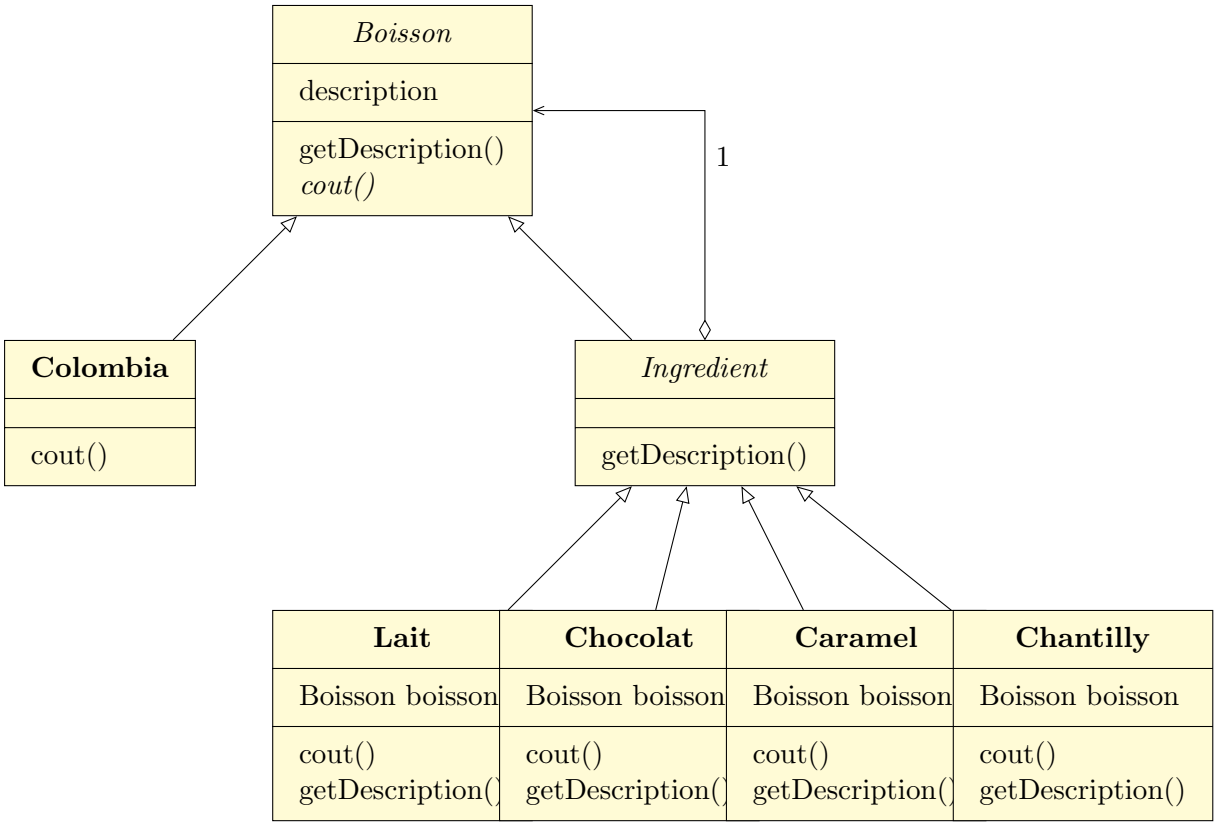
Question 1. *Une toute nouvelle chaîne de cafés se développe très rapidement. Elle décide de mettre à jour ses systèmes de prise de commandes afin de l'adapter à son offre. En plus du café, elle peut également proposer plusieurs ingrédients comme de la mousse de lait, du caramel, du chocolat, du sirop de vanille ou de noisette et couronner le tout de crème Chantilly. La chaîne facturant chacun de ces suppléments, elle a vraiment besoin de les intégrer à son système de commande.* Le diagramme ci-dessous représente la conception actuelle. Elle ne peut pas être changée car d'autres classes en dépendent.



Proposer une première solution simple à ce problème. Ecrire la méthode `double cout()` pour les classes `Boisson` et `Sumatra` en choissant des prix arbitraires.

Question 2. *L'augmentation du prix des ingrédients nous obligera à modifier le code existant. De nouveaux ingrédients nous forceront à ajouter de nouvelles méthodes et à*

modifier la méthode `cout()` *dans la superclasse. Que se passe-t-il si le client veut un double supplément de chocolat ?* Utiliser le design pattern Décorateur : écrire les classes du diagramme ci-dessous. Adapter les attributs et opérations des ingrédients à votre convenance, si nécessaire.



Question 2. Tester la commande de boissons.

```
public static void main(String[] args) {
        Boisson boisson = new Espresso();
        System.out.println(boisson.getDescription()
        + " : " + boisson.cout() + euros);

        Boisson boisson2 = new Sumatra();
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chantilly(boisson2);
        System.out.println(boisson2.getDescription()
        + " : " + boisson2.cout() + euros);
}
```

Question 3. *La chaîne décide de proposer trois tailles de boissons (petit, moyen et grand). La chaîne a vu cela comme une partie intrinsèque de la classe* `Boisson` *à laquelle elle a ajouté deux méthodes* `setTaille()` *et* `getTaille()`*. Le coût des suppléments dépend de la taille de la boisson.* Modifier le code en conséquent et adapter le test pour l'illustrer.

Question bonus. Permettre à l'utilisateur de choisir dynamiquement sa boisson, ses ingrédients et sa taille, puis afficher le prix.