TP

Elaboration d'un simulateur de canards

Partie 1

Ci-dessous le code initial :

```
public interface Cancaneur {
	public void cancaner();
}
public class Colvert implements Cancaneur {
	public void cancaner() {
		System.out.println("Coincoin");
	}
}
public class Mandarin implements Cancaneur {
	public void cancaner() {
		System.out.println("Coincoin");
	}
}
public class Appelant implements Cancaneur {
	public void cancaner() {
		System.out.println("Couincouin");

	}
}
public class CanardEnPlastique implements Cancaneur {
	public void cancaner() {
		System.out.println("Couic");
	}
}
```

Et voici le code de la classe principale :

```
public class SimulateurDeCanards {

	public static void main(String[] args) {
		SimulateurDeCanards simulateur = new SimulateurDeCanards();
		simulateur.simuler();
	}
	public void simuler() {
		Cancaneur colvert = new Colvert();
		Cancaneur mandarin = new Mandarin();
		Cancaneur appelant = new Appelant();
		Cancaneur canardEnPlastique = new CanardEnPlastique();
		System.out.println("\nSimulateur de canards");
		simuler(colvert);
		simuler(mandarin);
		simuler(appelant);
		simuler(canardEnPlastique);
	}
	public void simuler(Cancaneur canard) {
		canard.cancaner();
	}
}
```

1. Modélisez par un diagramme de classe cette conception actuelle du simulateur.

Partie 2

Nous voulons aussi traiter les oies bien qu'elles n'implémentent pas l'interface précédente.

```
public class Oie {
    public void cacarder() {
       System.out.println("Ouinc");
    }
}
```

2. Appliquez le patron « Adaptateur » à niveau objet en complétant le code ci-dessous

```
public class AdaptateurDOie ........... {
      ...................

      public AdaptateurDOie(..........) {
            .........;
      }
      public void cancaner() {
            ...........
      }
}
```

3. Donnez le diagramme de classe UML modélisant l'application de ce patron.

4. Complétez le code additionnel suivant du simulateur :

```
.......... canardOie = ..............
System.out.println("\nSimulateur de canards");
 [..]
 simuler(canardOie);
 [..]
```

5. Illustrez par un diagramme de séquence UML les échanges entre objets lors de l'appel à « simuler(canardOie) »

Partie 3

On souhaiterait connaître le nombre de couacs que fait une troupe de canards mais comment pouvons-nous ajouter la capacité de compter les couacs sans modifier les classes Canard ?

Nous allons pour cela appliquer le patron « décorateur » qui consistera ici à encapsuler un Cancaneur dans un objet là où on les utilise. Ensuite, lorsque l'objet décorateur recevra un appel à la méthode « cancaner » il déléguera le comportement au « cancaner » du cancaneur et comptera ensuite un couac de plus dans une variable de classe.

6. Complétez le code d'application suivant :

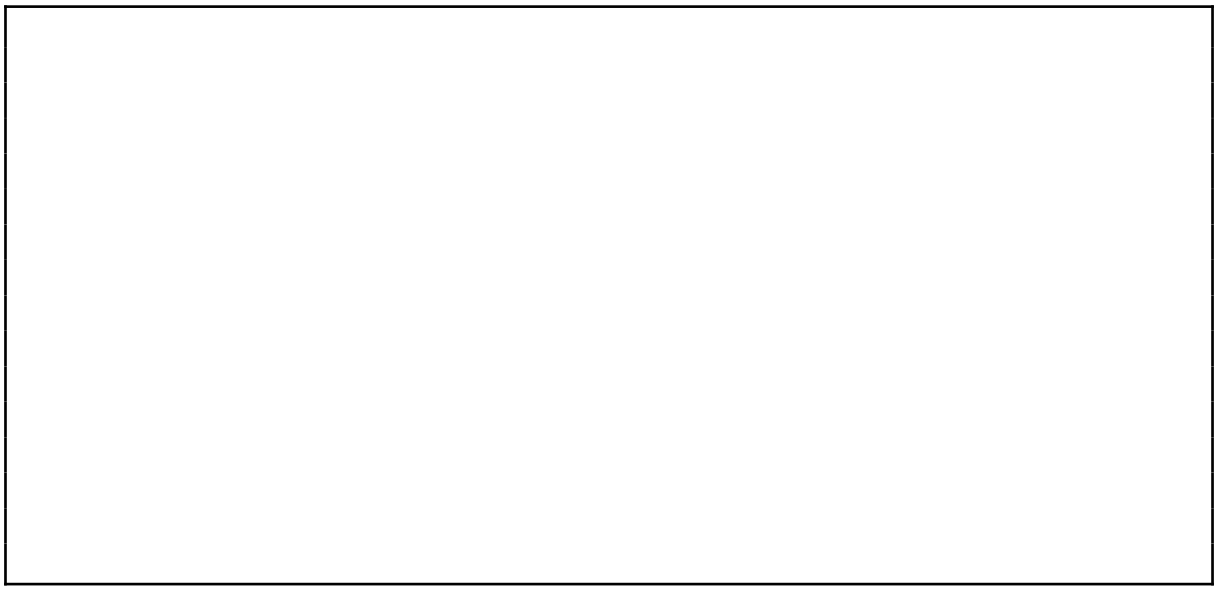
```
public class CompteurDeCouacs ..................... {
      private .......   ......;
      private ......  ....  nombreDeCouacs;

      public CompteurDeCouacs (.......) {
            this. ….....  = ….........;
      }
      public void cancaner() {
            ..................;
            nombreDeCouacs++;
      }
      public ...... .... getCouacs() {
            return nombreDeCouacs;
      }
}
```

7. Modifiez la création des objets pour utiliser ce décorateur et ajouter l'affichage du résultats de comptage :

```
      [..]
      Cancaneur colvert = ..................................
      simuler(colvert) ;
      [..]
      System.out.println("Nous avons compté " + ….............................
            + " couacs") ;
```

8. Donnez le diagramme de classe UML modélisant l'application de ce patron.



9. Illustrez par un diagramme de séquence UML les échanges entre objets lors de l'appel à « simuler(colvert) » (vous noterez que **décorateur** et **adaptateur** ont une logique similaire!)

Partie 4

Maintenant il nous faut une fabrique pour produire des canards. Un peu de contrôle qualité nous permettrait d'être certains que nos canards sont bien enveloppés. Nous allons construire une fabrique uniquement destinée à les créer. Comme la fabrique devra produire une famille de produits composée de différents types de canards, nous allons employer le pattern *Fabrique abstraite.*

10. Commençons par la définition de la FabriqueDeCanardsAbstraite :

```
public ..... class FabriqueDeCanardsAbstraite {
      public .....  ..... creerColvert();
      public .....  ..... creerMandarin();
      [..]
}
```

11. Complétez les 2 classes concrètes de fabriques suivantes :

```
public class FabriqueDeCanards ..................................... {
      public ..... creerColvert() {
            ..............................
      }
      public ..... creerMandarin() {
            ..............................
      }
}
public class FabriqueDeComptage ................................... {
      public ..... creerColvert() {
            ..............................................
      }
      public ..... creerMandarin() {
            ..............................................
      }
}
```

12. Complétez le code de tests suivant utilisant la fabrique de comptage :

```
public class SimulateurDeCanards {
      public static void main(String[] args) {
        SimulateurDeCanards simulateur = new SimulateurDeCanards();
        ..................... fabriqueDeCanards = ..............................
        simulateur.simuler(fabriqueDeCanards);
      }
      void simuler(.............................. fabriqueDeCanards) {
        ........... colvert = ..................................
        ........... mandarin = ...................................
        simuler(colvert);
        simuler(mandarin);
      }
      void simuler(Cancaneur canard) {
            canard.cancaner();
      }
}
```

13. Donnez le diagramme de classe UML modélisant l'application de ce patron.

Partie 5

Nous aimerions maintenant gérer des collections, et même des sous-collections de canards. Ce serait également agréable de pouvoir appliquer des opérations à tout l'ensemble de canards plutôt que de devoir les simuler un par un.

14. Quel est le pattern qui peut nous aider ?

15. Complétez le code de Troupe :

```
public class Troupe ……..  {

	private ……...

	public void add(……..)  {
		……..
	}
	public void …….	()  {
		…….
		…….
		…….
		…….
		…….
	}
```

Voici le code d'utilisation de la classe Troupe :

```
[..]
		Troupe troupeDeCanards = new Troupe();
		troupeDeCanards.add(mandarin);
		troupeDeCanards.add(appelant);

		Troupe troupeDeColverts = new Troupe();
		Cancaneur colvert2 = new Colvert();
		Cancaneur colvert3 = new Colvert();
		troupeDeColverts.add(colvert2);
		troupeDeColverts.add(colvert3);

		troupeDeCanards.add(troupeDeColverts);
		System.out.println("\nSimulateur de canards : Toute la troupe");
		simuler(troupeDeCanards);
		System.out.println("\nSimulateur de canards : Troupe de colverts");
		simuler(troupeDeColverts);
[..]
```

16. Donnez le diagramme de classe UML modélisant l'application de ce patron.

17. Illustrez par un diagramme de séquence UML les échanges entre objets lors de l'appel à « simuler(troupeDeCanards) »

Partie 6

Un **Cancanologue** aimerait maintenant observer le comportement des canards individuels. Il nous faut donc utiliser le patron *Observer*.

Voici l'interface que devront implémenter tous les cancaneurs pour pouvoir être observés :

```
public interface CouacObservable {
	public void enregistrerObservateur(Observateur observateur);
	public void notifierObservateurs();
	}
```

et celle d'un observateur :

```
public interface Observateur {
	public void actualiser(CouacObservable canard);
}
```

18. Complétez alors la définition de l'interface Cancaneur pour qu'ils soient tous observables :

```
public interface Cancaneur .................................{
	public void cancaner();
}
```

Maintenant, nous devons faire en sorte que toutes les classes concrètes qui implémentent Cancaneur puissent être un CouacObservable. Pour ce faire, nous pourrions implémenter l'enregistrement et la notification dans chaque classe mais nous préférons cette fois procéder un peu différemment : nous allons encapsuler l'enregistrement et la notification dans une autre classe, **Observable**, et la *composer* avec un **CouacObservable.** Ainsi, nous n'écrirons le code réel qu'une seule fois et CouacObservable n'aura besoin que du code nécessaire pour déléguer à la classe auxiliaire, Observable.

19. Complétez le code de la classe auxiliaire Observable :

```
public class Observable ..............................   {
      private …..... observateurs = new ..............();
      private ............ ..... canard;

      public Observable(.................. canard) {
            ..............................
      }
      public void enregistrerObservateur(..............................) {
            ..............................
      }
      public void notifierObservateurs() {
            .......... iterateur = ...................
            while (.....................) {
                  ..............................
                  ..............................
            }
      }
}
```

20. Intégrez l'auxiliaire Observable avec les classes Cancaneur sur l'exemple de l'implémentation de Colvert suivant : le colvert est un cancaneur qui est un CouacObservable **par transitivé :** il doit donc implémenter les méthodes de cette dernière interface.

```
public class Colvert ....................... {
      private ...................
      public Colvert() {
            ........... = new Observable(..........);
      }
      public void cancaner() {
            System.out.println("Coincoin");
            ........................
      }

      public void enregistrerObservateur(Observateur observateur) {
            ..............................................
      }
      public void notifierObservateurs() {
            ..............................................
      }

}
```

Remarquez que le code pour les autres canards de base (Mandarin, Appelant, CanardEnPlastique...) est identique.

En revanche, ceux de Troupe/ AdaptateurDOie / CompteurDeCouac seront différents (non traités ici).

21. Il ne reste plus qu'à gérer les observateurs. Complétez cette classe :

```
public class Cancanologue ......................{
      public void ..................(..................... canard) {
         System.out.println("Cancanologue : " + canard + " vient de cancaner.");
      }
}
```

22. Donnez le diagramme de classe UML modélisant l'application de ce patron (ne représentez que Colvert comme exemple de Cancaneur observable).

Voici un extrait de code pour notre simulateur :

```
[..]
          FabriqueDeCanardsAbstraite fabriqueDeCanards = new FabriqueDeCanards();
            Cancaneur colvert = fabriqueDeCanards.creerColvert();

            Cancanologue leCancanologue = new Cancanologue();

            colvert.enregistrerObservateur(leCancanologue);

            simuler(colvert);
[..]
```

23. Illustrez par un diagramme de séquence UML la notification des observateurs lors de l'appel à « simuler(colvert) »

Partie 7

Nous aimerions ajouter de nouveaux comportements à l'ensemble de nos cancaneurs (colvert / mandarin/ appelant / canard en plastique / adaptateur d'oie / troupe / compteur de couacs). Le problème c'est que nous ne connaissons pas forcément encore tous ces comportements…

Le patron *visiteur* nous permet de réaliser ceci. Les classes pourront accepter de nouveaux comportements à condition qu'elles acceptent d'être *visitées*. Les *visiteurs* seront des objets qui seront dédiés à un nouveau comportement. Ils possèderont une méthode d'action *visit* spécifique à chaque type d'objet *visitable* : cela leur permettra dans notre cas de séparer le comportement attendu pour chaque type de cancaneur.

Nous allons donc modifier toutes nos classes (une toute dernière fois) pour permettre l'ajout futur de nouveaux comportementaux qui n'auront plus besoin de les modifier.

Voici l'interface que devront implémenter tous les cancaneurs pour pouvoir être visités :

```
public interface Visitable {
	public void accept(Visitor v);
}
```

et celle d'un objet visiteur :

```
public interface Visitor {
	public void visit(Troupe t);
	public void visit(Colvert c);
	public void visit(Mandarin c);
	public void visit(Appelant c);
	public void visit(CanardEnPlastique c);
	public void visit(AdaptateurDOie c);
	public void visit(CompteurDeCouacs c);
	public void visit(Visitable c);
}
```

Remarquez bien qu'il y a une méthode *visit* par type de cancaneur connu et une dernière méthode permettant de capturer tout appel sur un *visitable* non cancaneur.

24. Complétez alors la définition de l'interface Cancaneur pour qu'ils soient tous visitables :

```
public interface Cancaneur .....................................{
      public void cancaner();
}
```

Maintenant, nous devons faire en sorte que toutes les classes concrètes qui implémentent Cancaneur puissent être *visitables*. Pour ce faire, nous allons considérer les types de cancaneurs simples (Colvert / Mandarin / Appelant / CanardEnPlastique / AdaptateurDOie) de ceux plus complexes (Troupe / CompteursDeCouacs).

25. Complétez le code de la méthode *accept* de tous les cancaneurs simples pour qu'elle ne fasse qu'un simple appel à la méthode *visit* du *visiteur* reçu en paramètre :

```
      …
      public void accept(Visitor visitor){
            ……….
      }
      …
```

26. Complétez le code de la méthode *accept* des objets **CompteursDeCouacs** pour qu'ils demandent au *visiteur* de les *visiter* mais qu'ils demandent également au cancaneur qu'ils décorent *d'accepter* le *visiteur* :

```
      …
      public void accept(Visitor visitor){
            ……….
            ……..
      }
      …
```

27. Complétez le code de la méthode *accept* des objets **Troupe** pour qu'ils demandent au *visiteur* de les *visiter* mais qu'ils demandent également à tous les cancaneurs qu'ils contiennent *d'accepter* le *visiteur* :

```
      …
      public void accept(Visitor visitor){
            ……….
             for (…..                )
                   ……..
      }
      …
```

28. Il ne reste plus qu'à gérer un objet *visiteur* pour donner un exemple d'ajout de comportement. Complétez le code de ce visiteur consistant à compter séparément les nœuds (objets Troupe et CompteursDeCouacs) des feuilles (les autres cancaneurs) d'une arborescence complexe de différents cancaneurs :

```
public class VisitorCompteurNoeudFeuille ……...{
      // les compteurs
      ……...
      ……...
      // les accesseurs aux compteurs
      public int getNbNoeud(){
            ……..
      }
```

```
	public int getNbFeuille(){
		……….
	}
	// les différentes méthodes visit
	public void visit(Troupe t) {
		……..
	}
	public void visit(Colvert c) {
		……..
	}
	public void visit(Mandarin c) {
		……..
	}
	public void visit(Appelant c) {
		……..
	}
	public void visit(CanardEnPlastique c) {
		……..
	}
	public void visit(AdaptateurDOie c) {
		……..
	}
	public void visit(CompteurDeCouacs c) {
		……..
	}
	public void visit(Visitable c) {
		System.out.println("classe pas encore gérée");
	}
}
```

Voici un extrait de code pour notre simulateur qui DOIT vous permettre de compter **4** nœuds et **6** feuilles :

```
[..]
public void simuler() {
	FabriqueDeCanardsAbstraite fabriqueDeCanards = new FabriqueDeComptage();
	Cancaneur colvert = fabriqueDeCanards.creerColvert();
	Cancaneur mandarin = fabriqueDeCanards.creerMandarin();
	Cancaneur canardOie = new AdaptateurDOie(new Oie());

	Troupe sousTroupe = new Troupe();
	fabriqueDeCanards = new FabriqueDeCanards();
	sousTroupe.add(fabriqueDeCanards.creerColvert());
	sousTroupe.add(fabriqueDeCanards.creerMandarin());
	sousTroupe.add(fabriqueDeCanards.creerColvert());

	Troupe maTroupe = new Troupe();
	maTroupe.add(colvert);
	maTroupe.add(mandarin);
	maTroupe.add(canardOie);
	maTroupe.add(sousTroupe);

	Visitor unVisiteur = new VisitorCompteurNoeudFeuille();
	maTroupe.accept(unVisiteur);
	System.out.println("Nb de noeuds  = "
			+ ((VisitorCompteurNoeudFeuille) unVisiteur).getNbNoeud());
	System.out.println("Nb de feuilles  = "
			+ ((VisitorCompteurNoeudFeuille) unVisiteur).getNbFeuille());
	}
[..]
```

29. Illustrez par un diagramme de séquence UML **partiel** les échanges de messages entre cancaneurs complexes et simples suite à l'appel à `maTroupe.accept(unVisiteur);`