CPOA - Subject TD 1

Dut/Info-S3/M3105 - (Week 45)

PreReq	1. I know how to code in Java. 2. I know I need to think before I start coding. 3. I know basic OO concepts (inheritance, polyporphism, …).
ObjTD	Understand the importance of **Design**.
Duration	**1** TD and **2** TPs (spread on 2 weeks)

1. The "SuperCanard" application

This TD exercice is inspired from the excellent book "Head First: Design Pattern". Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.

1.1. Existing application

You are asked to work on an existing app `SuperCanard` (duck, called *canard* in French, simulation game) which model (sorry for the French) is provided in the following class diagram:

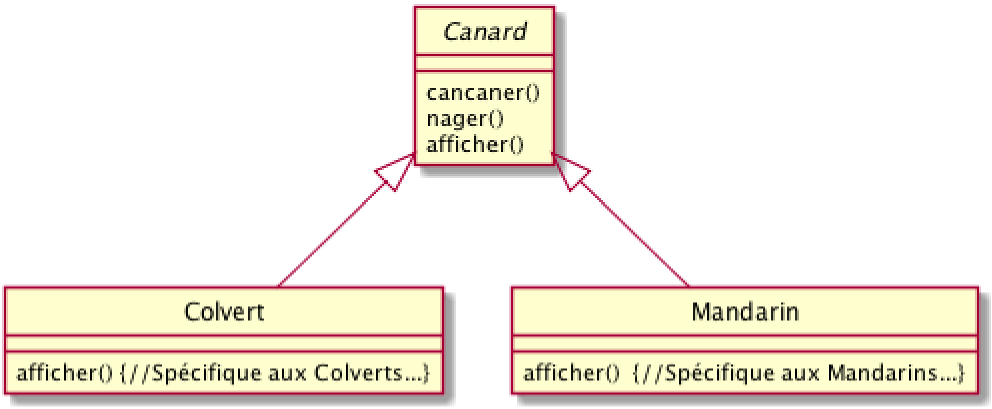


Figure 1. Existing app model (plantUML source here)

Some other classes inherit from `Canard`.

Here is a code example:

First version of `Canard.java`

```
abstract public class Canard {

    public void cancaner() {
        System.out.println("Je cancane comme un Canard!");
    }

    public void nager() {
        System.out.println("Je nage comme un Canard!");
    }

    abstract public void afficher();
}
```

First version of `Colvert.java`

```
public class Colvert extends Canard {

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

}
```

1.2. Modification/Improvement

Your boss requires that you upgrade the application in order to be a little more realistic.

You decide to add a `voler()` method to all your ducks:

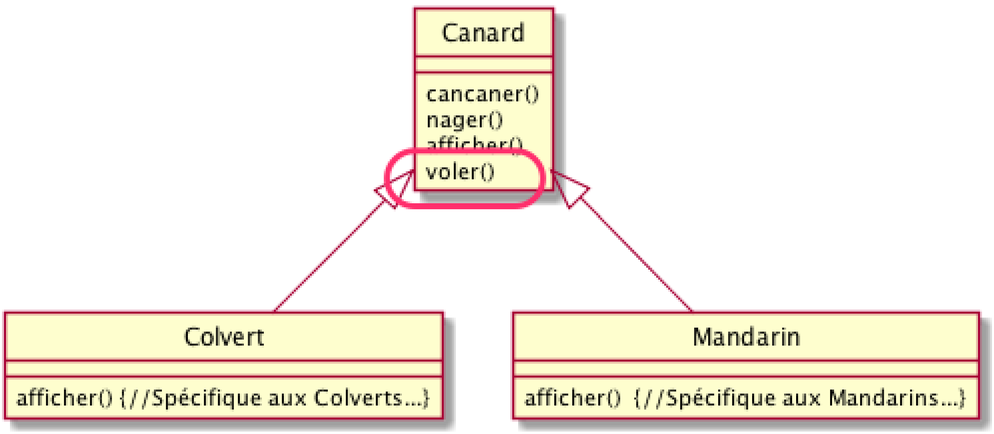


Figure 2. New feature

Second version of `Canard.java`

```
abstract public class Canard {

    public void cancaner() {
        System.out.println("Je cancane comme un Canard!");
    }

    public void nager() {
        System.out.println("Je nage comme un Canard!");
    }

    abstract public void afficher();

    public void voler() {
        System.out.println("Je vole comme un Canard!");
    };
}
```

1.3. #WTF!

You receive an emergency call from your boss: in the application some plastic ducks start to fly!!! In addition, sick ducks, that shouldn't fly, do so!

You forgot that some kind of ducks do not fly!

QUESTION

Complete this sentence: **Inheritance** is great to do .... but is more problematic in terms of ....

1.4. Solution 1: redefine the methods

The first solution that comes to your mind is simple: redefine the `voler()` method for the ducks who don't fly.

QUESTION

Complete the following java code to implement this solution:

```
public class CanardEnPlastique extends Canard {

    @Override
    public void afficher() {
        System.out.println("Je suis un CanardEnPlastique!");
    }






}
```

QUESTION

In the following list, what are the problems that inheritance can raise to define the behavior of a `Canard`? (Possibly mutliple good answers) :

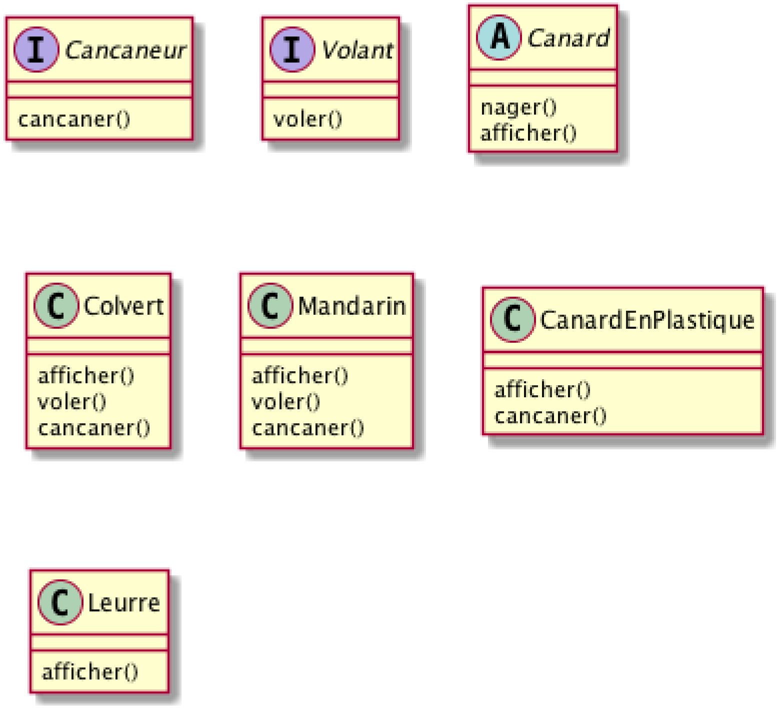
- ▢ Code is duplicated (rewritten) between sub-classes.
- ▢ Behavior changes at run-time are complicated.
- ▢ We cannot have dancing ducks.
- ▢ It is hard to know all the ducks' behaviors
- ▢ Ducks cannot fly and sing at same time.
- ▢ Modifications can modify unexpectedly other ducks' behavior.

1.5. Solution 2: use of interfaces

You know try the use of *interfaces* to improve the code.

QUESTION

1. On the following diagram, place the inheritance relations (jave `extends`) and the implementation relations (java `implements`):



2. What do you think of the final result ?

1.6. Solution 3: isolate what change

You realize you're facing the kind of problem you had in the `MPA` module: **CHANGES**!

Let us then apply our first *good principle* :

Good design principle

Identify aspects of your code that vary and separate them from the one that don't.

QUESTION

What are the two main things that vary in your code?

1.6.1. Implementing behaviors

Let's try to implement behavior differently, so that they are separated from the rest of the code. For that we will use another good principle:

Good design principle

Program an interface, not an implementation.

QUESTION

Propose a design (class diagram only) with the following classes and/or interfaces (you'll have to decide): `ComportementVol`, `VolerAvecDesAiles`, `NePasVoler`.

1.6.2. Adding the new behaviors to the code

We have now to somehow link the behaviors to their corresponding ducks' class.

QUESTION

1. Add to the `Canard` class two attributes to reference their behaviors.
2. Remove the useless methods.
3. Replace them (provide the corresponding code) by the methods: `effectuerVol()` and `effectuerCancan()` (using the new attributes).
4. Modify the constructors of `Colvert` (for example) how the attributes are initialized.
5. Add to `Colvert` the `setMalade()` and `setGueri()` methods that allow at run-time to modify the flying behavior.

1.6.3. Summary and discussions

Let us now have a look at the overall design we have obtained.

QUESTION

1. Draw the class diagram of the new application.
2. What would you modify to add for example a new flying mode (e.g., *propulsion à réaction*) ?
3. Could you thhink of a class that could use the `Cancan` behavior without being a duck?

1.7. Your first *Design Pattern*

1.7.1. The Strategy pattern

In fact you have just implemented your first *Design Pattern* : the *Strategy* pattern (**Stratégie**), sorry for the French:

Design pattern: ***Stratégie*** *(Strategy)*

Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

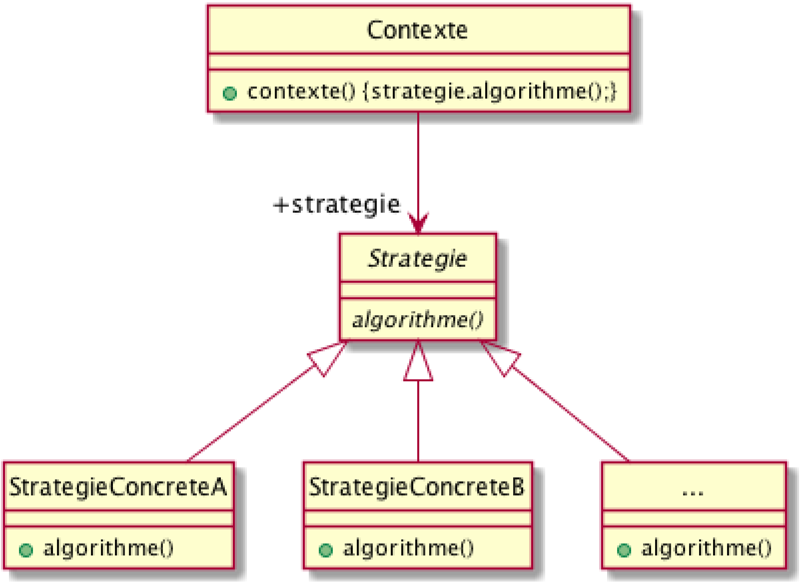


Figure 3. Modèle UML du patron Strategy

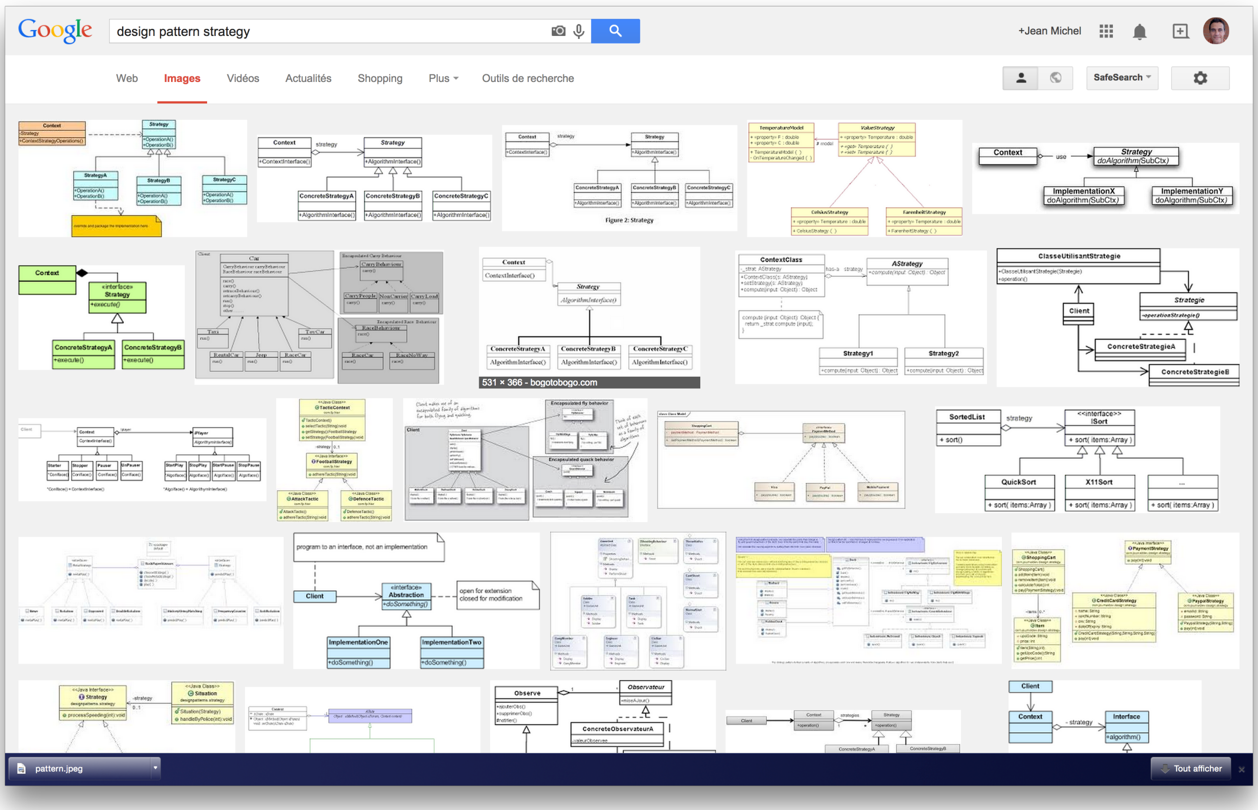
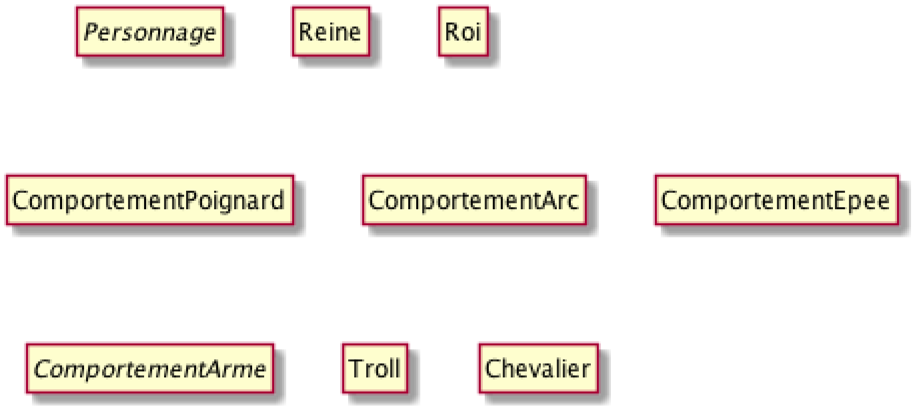


Figure 4. Some examples of descriptions of Strategy

1.7.2. Let's try it on another application

You are asked to rework on an application where only the following model was produced (sorry again for the damn French):



1. Reorganize the classes
2. Identify abstract classes, interfaces and regular classes.
3. Trace the links between classes ("is a", implementation, "has a")
4. Place the following `setArme()` method on the correct class:

```
setArme(ComportementArme a) {
  this.arme = a;
}
```

Still hungry?

We have used without mentioning it a 3rd good principle:

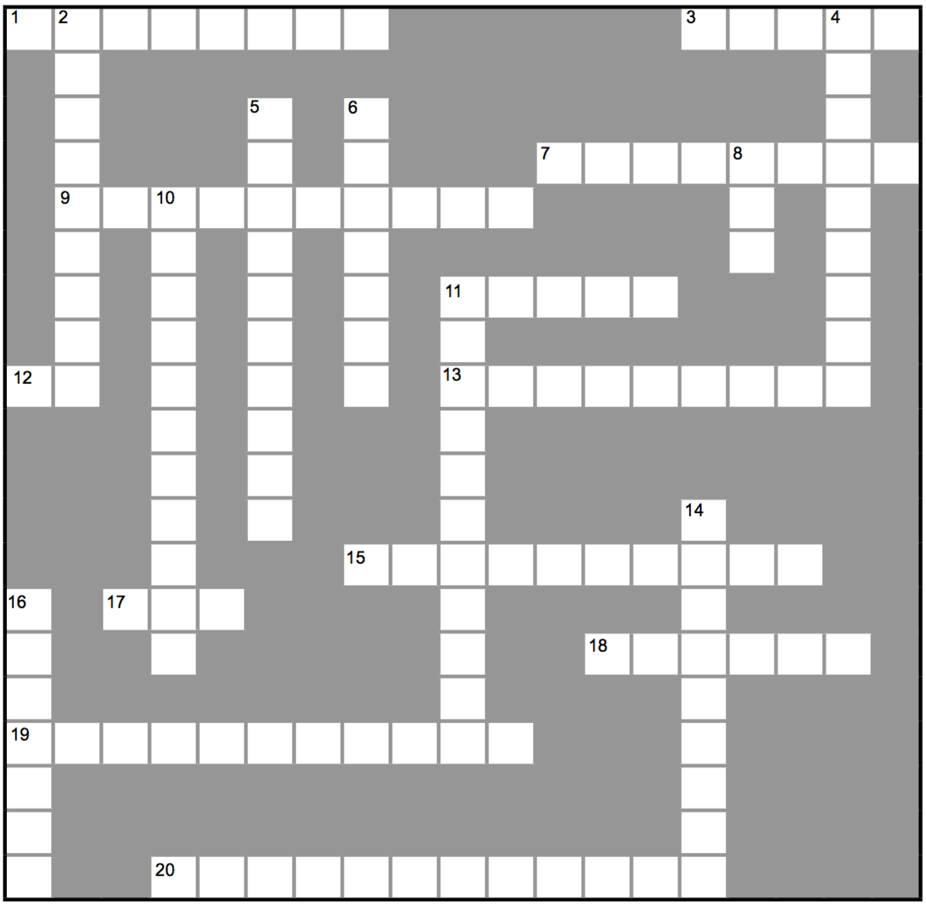
Good design principle

Prefer **composition** than **inheritance**.

QUESTION

What difference is there between our final design and this kind of implementation?:

```
abstract public class Canard implements ComportementVol {...}
```



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.

Figure 5. **CrossWords** *(sorry, in French)*

> This crossword is taken from the book mentioned earlier, and hence includes definitions of words that you can't guess:

- 7 ⇒ Baleares
- 11 ⇒ Observateur
- 12 ⇒ DK

> **QUESTION**
>
> How would you test the presence of a Strategy pattern in an implementation ?

Do not hesitate to have a look at this other *role playing* example, available here (p.116).