TD 6 - Principes SOLID

Exercice 1. Single Responsability Principle

Pourquoi le code qui suit ne respecte pas ce principe ? Comment le corriger ?

```
public class PayRollService {
    ...

    public void payEmployee (Employee employee) {
        if (employee.getType() == EmployeeType.PERMANENT ) {
            ...
        } else if (employee.getType() == EmployeeType.CONSULTANT) {
            ...
        }
        paycheck = checkService.issuePaychek(employee);
        this.sendPaycheck(paycheck);
    }
}
```

Solution :

Ici il y a deux raisons de changer : 1) lorsque la logique de paiement selon les types d'employé changera et 2) lorsque la logique d'émission/envoi des chèques changera. La classe `Employee` devrait elle-même gérer la logique qui dépend de son type. La méthode `sendPaycheck` de la classe pourrait également être déplacée.

Exercice 2. Open/Closed Principle

En quoi le code qui suit respecte-t'il ce principe, et en quoi le viole-t'il ?

```
public class Vehicle {
  public void startVehicle() { ... }
}

public class Car extends Vehicle {
  @Override
  public void startVehicle() {
    super.startVehicle();
  }
}

public class Truck extends Vehicle {
  @Override
  public void startVehicle() {
    super.startVehicle();
  }
}

public class GasStation {
  public void refillVehicle(Vehicle vehicle) {
    if(vehicle instanceof Car) { ... }
    else { ... }
  }
}
```

Solution :

Respect de l'OCP : Le design est flexible et l'ajout d'un nouveau type de véhicule pourra être fait simplement sans ajouter de modifications au code existant.
Non respecte de l'OCP : dans la classe `GasStation` l'ajout d'un nouveau type peut amener des modifications à la méthode `refillVehicle`.

Exercice 3. Liskov Substitution Principle

On modifie l'exercice précédent en modifiant la class `Truck` :

```
public class Truck extends Vehicle {
    @Override
    public void startVehicle() {
        this.startMileageTracker();
        super.startVehicle();
    }
  }
}
```

et en ajoutant la classe `PremiumGasStation` :

```
public class PremiumGasStation extends GasStation {
    @Override
    public void refillVehicle(Vehicle vehicle) {
        if(vehicle.isType(Car)) {
            throw new RefillCarException();
        }
        super.refillVehicle();
    }
    ...
}
```

Le principe de substitution est-il respecté ?

Solution :

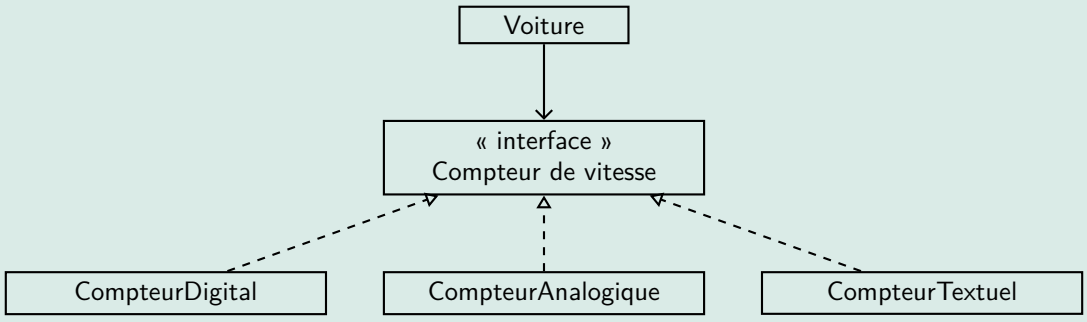
Ici il y a non respect du LSP pour car la classe enfant `PremiumGasStation` tient compte du type de véhicule pour appliquer une condition particulière que la méthode de sa classe mère n'a pas. Ce qui veut dire que si on remplaçait `GasStation` par `PremiumGasStation`, les clients qui utilisent `GasStation` seraient impactés.
À noter que l'ajout de `startMileageTracker` dans la méthode `startVehicle` du camion est légitime si on sait que cela ne brise pas le comportement principal et que c'est une règle attendue par les clients pour les camions.

Exercice 4. Interface Segregation Principle

Une voiture doit afficher sa vitesse sur 3 types d'affichage possibles en fonction du modèle de la voiture : digital, à aiguille ou en texte.

Concevez un système (un simple diagramme statique suffit) qui permet de répondre à ce besoin.

Solution :



Exercice 5. Dependency Inversion Principle

Question 1. - *Cognage de clous* - Créez un projet en respectant les critères suivants :

— Un marteau peut cogner un clou.

— Un clou ne peut se faire cogner qu'une seule fois.

— Lorsqu'un clou se fait cogner, il envoi un message à l'utilisateur disant qu'il a été cogné.

Solution :

Voici d'abord une **mauvaise** solution :

```java
public class Hammer {
    public void hit(Nail nail) {
        nail.hit();
    }}

public class Nail {
    private boolean isHit = false;

    public void hit() {
        if (!isHit) {
            isHit = true;
            System.out.println("Ouch!");
        }}}
```

Remarquez que l'écriture d'un message à l'utilisateur est un niveau plus bas que notre domaine (i.e. Marteau et Clou). Voilà maintenant une bonne solution :

```java
public class Hammer {
    public void hit(Nail nail) {
        nail.hit();
    }}

public class Nail {
    private final Notifier notifier;
    private boolean isHit = false;

    public Nail(Notifier notifier) {
        this.notifier = notifier;
    }

    public void hit() {
        if (!isHit) {
            isHit = true;
            notifier.doNotify("Ouch!");
        }}}

public interface Notifier {
    void doNotify(String message);
}

public class ConsoleNotifier implements Notifier {
    public void doNotify(String message) {
        System.out.println(message);
    }}
```

Afin d'éviter que `Nail` (haut niveau) connaisse le mode de communication avec l'utilisateur (bas niveau), nous utilisons une abstraction `Notifier`.

Question 2. - *Par Minou* - Dans un jeu où les lions prétendent être des chats afin de mieux les manger, créez un projet tout en respectant les critères suivants :

- Le jeu commence avec une quantité de 100 souris.
- Le jeu détient une liste de 8 chats et 2 lions. Chaque espèce a accès à une action différente :
  - Chat : Manger une souris. Ceci décrémente de 1 le nombre de souris du jeu.
  - Lion : Manger un chat. Ceci retire le chat en question de la liste des chats du jeu.
- Respectez le DIP.

Pour vous aider, nous vous fournissons deux classes que vous pouvez modifier comme vous voulez :

```java
public class Application {

  private static final int INITIAL_NUMBER_OF_MICE = 100;
  private static final int INITIAL_NUMBER_OF_CATS = 8;
  private static final int INITIAL_NUMBER_OF_LIONS = 2;

  public Application() {
    Game game = new Game(INITIAL_NUMBER_OF_MICE, INITIAL_NUMBER_OF_LIONS,
    INITIAL_NUMBER_OF_CATS);
  }
}

public class Game {

  private int numberOfMice;
  private final List<Cat> cats;
  private final List<Lion> lions;

  public Game(int numberOfMice, int numberOfLions, int numberOfCats) {
    this.numberOfMice = numberOfMice;
    this.cats = new ArrayList<>();
    this.lions = new ArrayList<>();

    for (int i = 0; i < numberOfLions; i++) {
      lions.add(new Lion());
    }

    for (int i = 0; i < numberOfCats; i++) {
      cats.add(new Cat());
    }
  }
}
```

Solution :

Pour respecter le DIP dans cet exercice, il faut permettre aux chats et aux lions (haut niveau) de modifier les éléments du jeu (bas niveau), sans leurs donner un accès direct.

```java
// Voici les deux interfaces utilisees:
public interface MouseEater {
    void eatAMouse();
}

public interface CatEater {
    void eatACat();
}

public class Cat {

    private final MouseEater mouseEater;

    public Cat(MouseEater mouseEater) {
        this.mouseEater = mouseEater;
    }

    public void eatAMouse() {
```

```
19          mouseEater.eatAMouse();
20      }
21  }
22
23
24  public class Lion {
25
26      private final CatEater catEater;
27
28      public Lion(CatEater catEater) {
29          this.catEater = catEater;
30      }
31
32      public void eatACat(Cat cat) {
33          catEater.eatACat(cat);
34      }
35  }
36
37  // Voici un exemple d'implementation, mais ca pourrait etre n'importe quoi:
38  public class CatList extends ArrayList<Cat> implements CatEater {
39
40      public void eatACat(Cat cat) {
41          this.remove(cat);
42      }
43  }
44
45  public class MousePopulation implements MouseEater {
46
47      private int numberOfMice;
48
49      public MousePopulation(int numberOfMice) {
50          this.numberOfMice = numberOfMice;
51      }
52
53      public void eatAMouse() {
54          numberOfMice--;
55      }
56  }
57
58  // Ces implementations sont ensuite ajoutees au jeu:
59  public class Game {
60
61      private final MousePopulation mousePopulation;
62      private final List<Lion> lions;
63      private final CatList cats;
64
65      public Game(int numberOfMice, int numberOfLions, int numberOfCats) {
66          this.mousePopulation = new MousePopulation(numberOfMice);
67          this.lions = new ArrayList<>();
68          this.cats = new CatList();
69
70          for (int i = 0; i < numberOfLions; i++) {
71              lions.add(new Lion(cats));
72          }
73
74          for (int i = 0; i < numberOfCats; i++) {
75              cats.add(new Cat(mousePopulation));
76          }
77      }
78  }
```

Dans le cadre de cet exercice, cette solution est suffisante. Cependant, il y a toujours place à amélioration.