OUAP 4104, Examen 2020, en distanciel

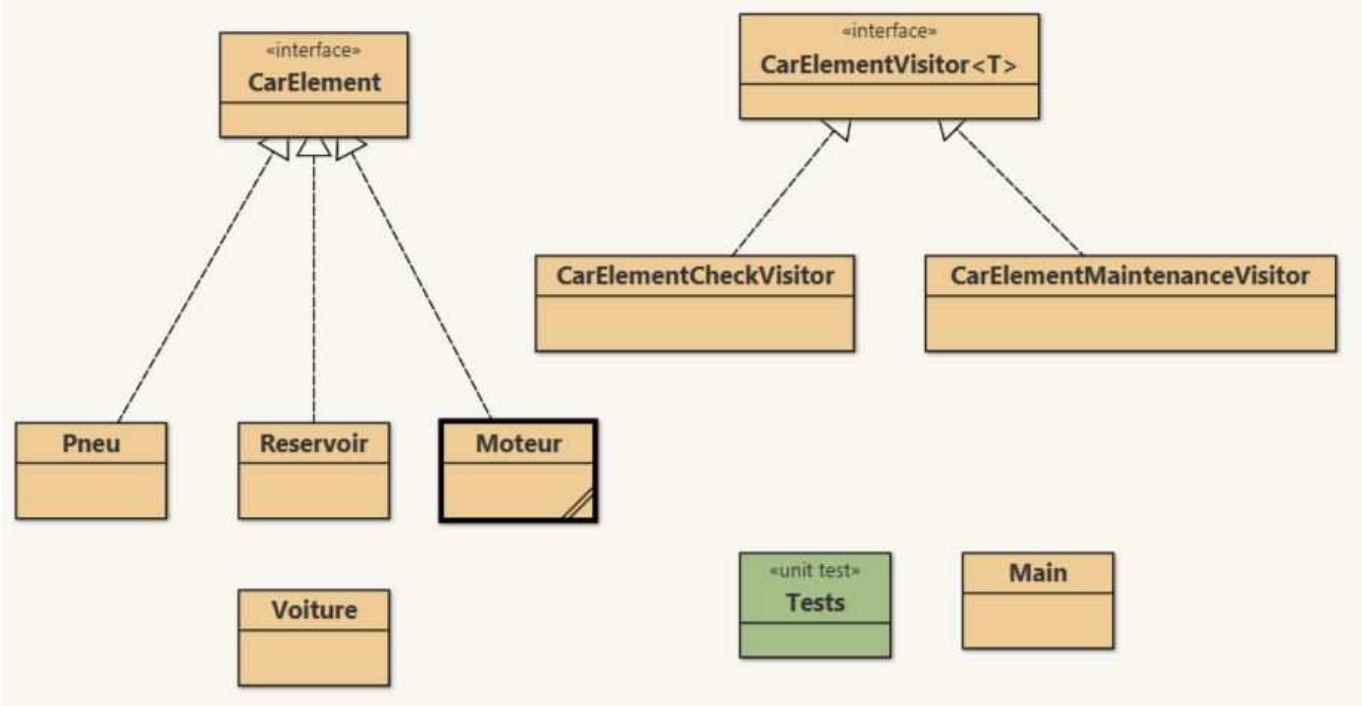
- Votre travail devra être déposé avant 20h30, toujours via Bluej item submit,
  attention à l'échéance de 20h30 l'outil n'a pas de tolérance...
- Et avant 20h30, générez l'archive java de votre travail sous Bluej menu Projet item Exporter/Create Jar File et envoyez moi cette archive via WeTransfer à jean-michel.douin@esiee.fr, en indiquant vos nom/prénom
- Attention tous vos rendus seront analysés avec JPlag

Question 1 – Patron Visiteur

Le patron Visiteur permet de séparer des données et les traitements associés pour ces données. Cela permet de mettre en place un traitement adapté aux détails publics des objets visés, de façon externe.
« *En pratique, le modèle de conception visiteur est réalisé de la façon suivante : chaque classe pouvant être « visitée » doit mettre à disposition une méthode publique « accepter » prenant comme argument un objet du type « visiteur ». La méthode « accepter » appellera la méthode « visite » de l'objet du type « visiteur » avec pour argument l'objet visité. De cette manière, un objet visiteur pourra connaître la référence de l'objet visité et appeler ses méthodes publiques pour obtenir les données nécessaires au traitement à effectuer (calcul, génération de rapport, etc.).* » (*wikipedia*)

Considérons une voiture vue à travers quelques composants qui doivent remplir certaines conditions de sécurité avant utilisation : les pneus doivent avoir une pression de 2,5 bars, l'huile moteur une pression de 3 bars et le réservoir d'essence doit être plein.
Ces composants implémentent l'interface CarElement.
L'interface Visiteur CarElementVisitor donne la signature des *visites*.

Voici le schéma complet des classes manipulées :



L'interface CarElement est fournie :

```
public interface CarElement {

	// Méthode à définir par les classes implémentant CarElements
	public <T> T accept(CarElementVisitor<T> visitor);

	// Vérifie que les élément sont en état de sécurité
	// (pression adequat ...)
	public boolean safetyCheck();

	// met les élément en état de sécurité s'ils ne le sont pas.
	public void setSafe();
}
```

… Ainsi que l'interface CarElementVisitor :

```
public interface CarElementVisitor<T> {
    public T visit(Pneu pneu);
    public T visit(Moteur moteur);
    public T visit(Reservoir reservoir);
    public T visitCar(Voiture voiture);
}
```

La classe Main …

```
public class Main {

static public void main(String[] args) {

Voiture voiture = new Voiture();

CarElementVisitor<String> checkVisitor = new CarElementCheckVisitor();
CarElementVisitor<String> maintenanceVisitor = new
	CarElementMaintenanceVisitor();

System.out.println(checkVisitor.visitCar(voiture));
System.out.println(maintenanceVisitor.visitCar(voiture));
	}
}
```

… déclenche l'affichage suivant :

```
	Compte rendu des niveaux du vehicule
	DANGER !!! Pression du pneu avant gauche : 1.8
	DANGER !!! Pression du pneu avant droit : 1.8
	DANGER !!! Pression du pneu arriere gauche : 1.8
	DANGER !!! Pression du pneu arriere droit : 1.8
	ATTENTION !!! Le reservoir n'est pas plein
	DANGER !!! Pression d'huile moteur : 2.0
	Niveaux du vehicule transmis

	Verification des niveaux du vehicule
	Pneu avant gauche regonfle. Pression : 2.5
	Pneu avant droit regonfle. Pression : 2.5
	Pneu arriere gauche regonfle. Pression : 2.5
	Pneu arriere droit regonfle. Pression : 2.5
	Le reservoir a ete rempli
	Ajout d'huile moteur. Pression d'huile: 3.0
```

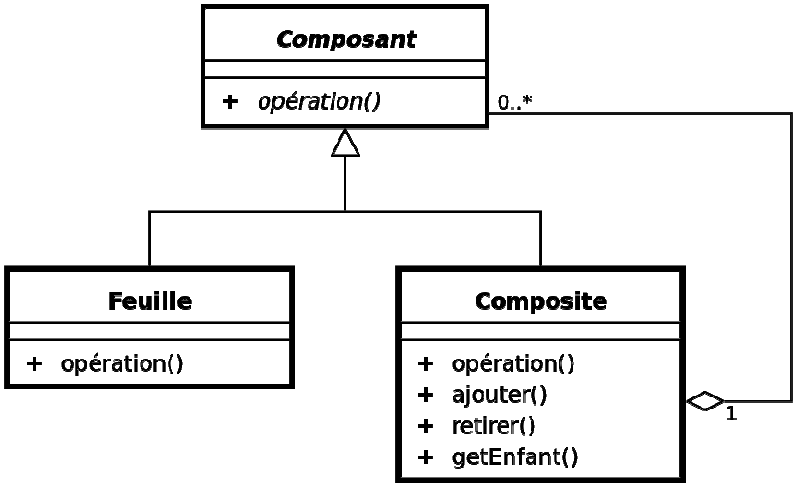
```
Niveaux du vehicule verifies
```

La classe de tests est à lire attentivement, et vous devez compléter les classes Pneu, Reservoir, Moteur, CarElementCheckVisitor et CarElementMaintenanceVisitor dont les trames sont fournies.

Question 2 – Patron Composite + visiteur

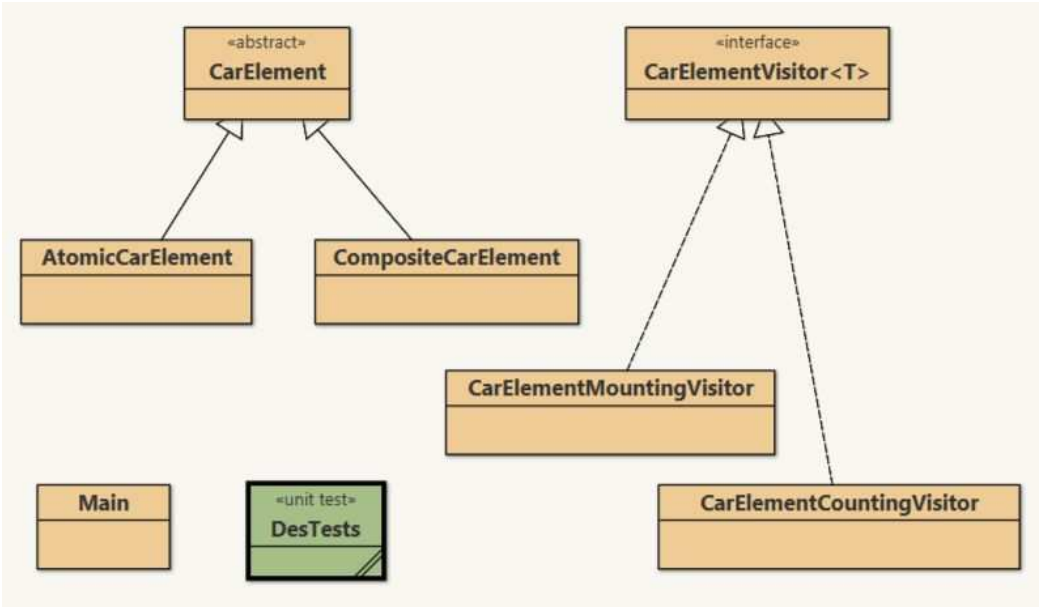
Pour ajouter de nouvelles manipulations à un Composite, on peut modifier le Composite lui-même, mais cela peut-être contraignant si on a beaucoup d'action à ajouter. Le Visiteur est alors une solution.
Pour mémoire, le patron Composite permet de concevoir une structure arborescente récursive permettant d'implémenter les feuilles et les composites avec la même interface logicielle, afin qu'ils soient manipulés de la même manière.

Voici, toujours pour rappel, un diagramme UML du patron Composite :



Reprenons l'exemple d'une voiture. Nous sommes maintenant sur une chaîne de montage. Doivent être assemblés le volant, les roues et la colonne de direction, qui forment le système de direction (les 4 roues formant elles-mêmes un ensemble « roues ») et le carburateur et les cylindres qui forment un ensemble moteur. Le système de direction et l'ensemble moteur forment une voiture.

Voici le schéma complet des classes manipulées :



La classe abstraite CarElement est fournie :

```java
public abstract class CarElement {

    private String name;
    private CarElement parent;

    public CarElement (String name) {
        this.name=name;
    }

    public abstract <T> T accept(CarElementVisitor<T> visitor);

    public final String getName() {
        return name;
    }

    public CarElement getParent(){
        return this.parent;
    }

    public void setParent(CarElement parent){
        this.parent = parent;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName()+" - "+this.getName();
    }
}
```

… ainsi que l'interface CarElementVisitor :

```java
public interface CarElementVisitor <T> {

    public T visit(AtomicCarElement element);
    public T visit(CompositeCarElement system);
}
```

La classe Main …

```
public class Main{

    static public void main(String[] args) throws Exception {

       AtomicCarElement volant = new AtomicCarElement("volant");
        AtomicCarElement colonne =  new AtomicCarElement("colonne de
direction");
        AtomicCarElement roue1 = new AtomicCarElement("roue avant droite");
        AtomicCarElement roue2 = new AtomicCarElement("roue avant gauche");
        AtomicCarElement roue3 = new AtomicCarElement("roue arriere gauche");
        AtomicCarElement roue4 = new AtomicCarElement("roue arriere droite");
        AtomicCarElement carbu = new AtomicCarElement("carburateur");
        AtomicCarElement cylindres = new AtomicCarElement("cylindres");

        CompositeCarElement roues = new CompositeCarElement("roues");

roues.addCarElement(roue1).addCarElement(roue2).addCarElement(roue3).addCarEleme
nt(roue4);
        CompositeCarElement direction = new CompositeCarElement("Systeme de
direction");

direction.addCarElement(volant).addCarElement(colonne).addCarElement(roues);
        CompositeCarElement moteur = new CompositeCarElement("moteur");
        moteur.addCarElement(carbu).addCarElement(cylindres);
        CompositeCarElement voiture = new CompositeCarElement("Porsche 911");
        voiture.addCarElement(direction).addCarElement(moteur);

        System.out.println(voiture.accept(new CarElementMountingVisitor()));
        System.out.println("Nombre d'elements dans ce vehicule :
"+voiture.accept(new CarElementCountingVisitor()));
    }
}
```

… déclenche l'affichage suivant :

«

```
      Demarrage du montage de l'ensemble Porsche 911
            Demarrage du montage de l'ensemble Systeme de direction
                  Montage de l'element : volant
                  Montage de l'element : colonne de direction
                  Demarrage du montage de l'ensemble roues
                        Montage de l'element : roue avant droite
                        Montage de l'element : roue avant gauche
                        Montage de l'element : roue arriere gauche
                        Montage de l'element : roue arriere droite
                  Fin du montage de l'ensemble roues
            Fin du montage de l'ensemble Systeme de direction
            Demarrage du montage de l'ensemble moteur
                  Montage de l'element : carburateur
                  Montage de l'element : cylindres
            Fin du montage de l'ensemble moteur
      Fin du montage de l'ensemble Porsche 911

      Nombre d'elements dans ce vehicule : 8
```

»

La classe de tests est à lire attentivement, vous devez compléter les classes AtomicCarElement, CompositeCarElement, CarElementMountingVisitor et CarElementCountingVisitor dont les trames sont fournies.

Question 3 – question de cours

Soit la classe ci-dessous dont l'implémentation des méthodes **pre** et **post** est laissée à la responsabilité des sous-classes.

a) Complétez la classe A :

```
public abstract class A {
  public void m(int i) throws Exception{
    pre(i>=0);
    i = i + 1 ;
    post(i>0);
  }
     // à compléter dans le paquetage question3
}
```

b) Proposez une sous-classe de A, nommée B permettant d'afficher la valeur du booléen transmis lors de l'appel de pre et post et de lever une exception (classe Exception) si l'évaluation de l'expression booléenne transmise est fausse (valeur false).

c) Donnez un exemple d'utilisation de cette classe : la classe Main par exemple

d) De quel patron s'agit-il ? il vous suffit de l'indiquer en commentaires dans la classe d'utilisation (Main)

- Votre travail devra être déposé avant 20h30, toujours via Bluej item submit,
    - attention à l'échéance de 20h30 l'outil n'a pas de tolérance...
- Et avant 20h30, générer l'archive java de votre travail sous Bluej menu Projet item Exporter/Create Jar File et envoyez moi cette archive via WeTransfer à jean-michel.douin@esiee.fr, en indiquant vos nom/prénom

Nous utiliserons le fameux outil JPlag afin d'effectuer une mesure précise sur vos rendus, cet outil est assez précis comme l'indique l'exemple en ligne

- https://jplag.ipd.kit.edu/ et https://jplag.ipd.kit.edu/example/index.html