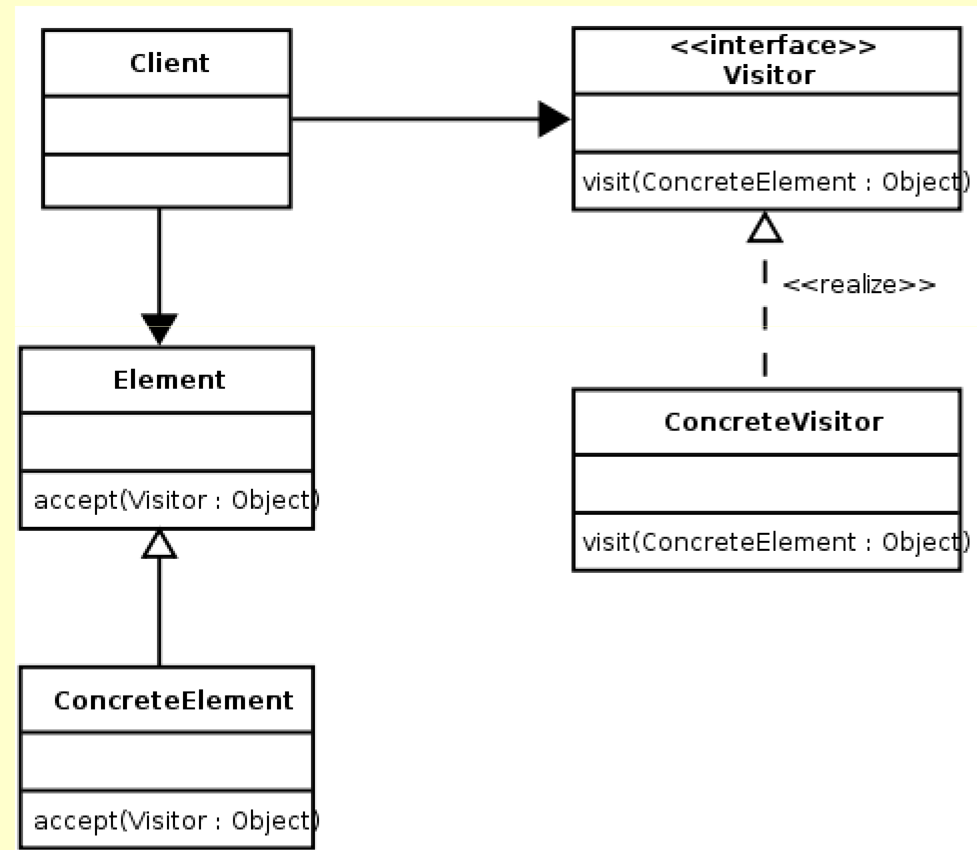
Patron: Visitor/Visiteur

Découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.

Un visiteur possède une méthode par type d'objet traité. Pour ajouter un nouveau traitement, il suffit de créer une nouvelle classe dérivée de la classe Visiteur.

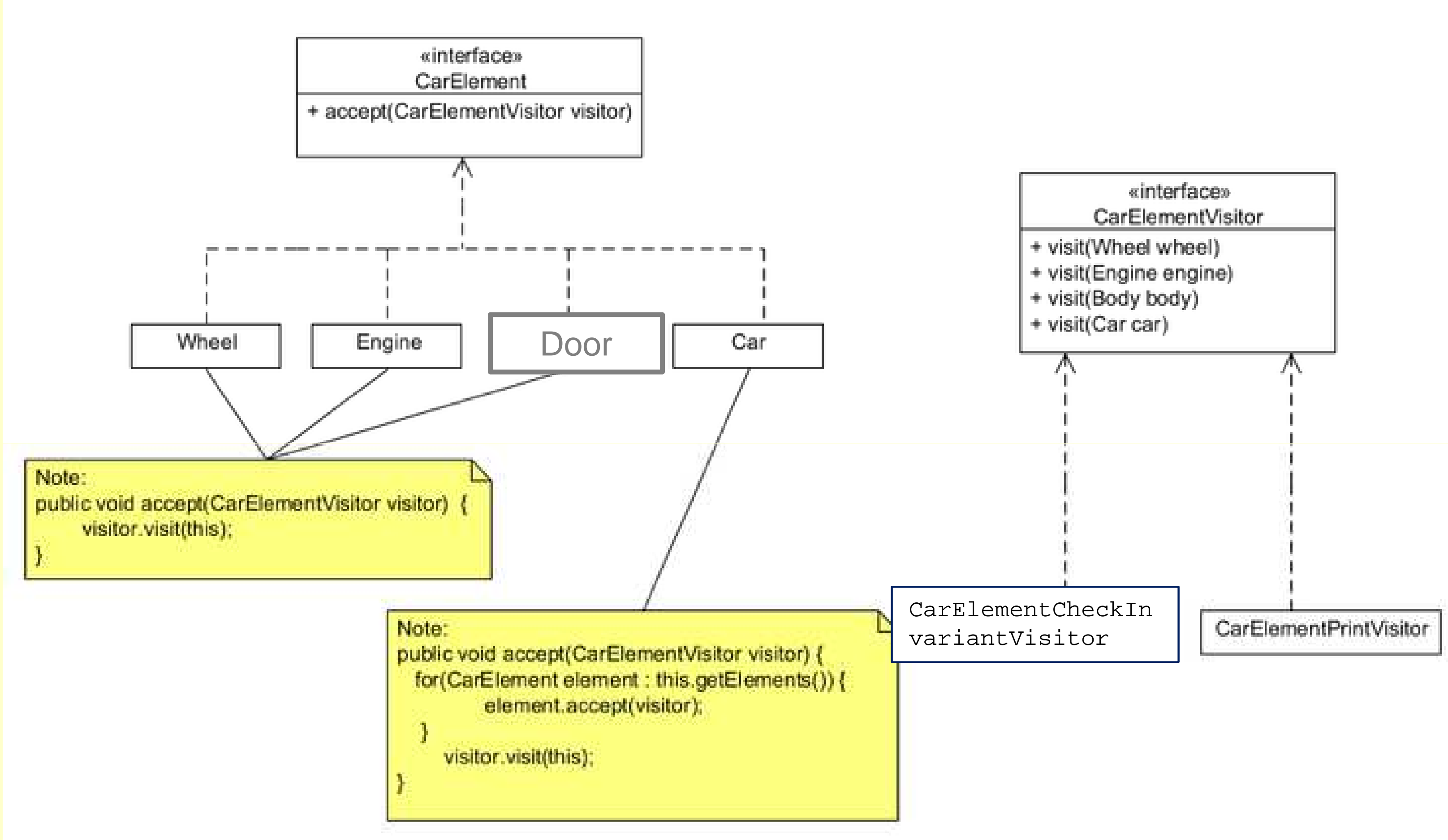


Java Exemple: Voiture/Car

L'exemple suivant montre comment afficher un arbre de nœuds (les composants d'une voiture).

Au lieu de créer des méthodes d'affichage pour chaque sous-classe (`Wheel, Engine, Door`, et `Car`), une seule classe est créée (`CarElementPrintVisitor`) pour afficher les éléments.

Parce que les différentes sous-classes requiert différentes actions pour s'afficher proprement, la classe `CarElementPrintVisitor` répartit l'action en fonction de la classe de l'argument qu'on lui passe.


«interface»
CarElement
+ accept(CarElementVisitor visitor)
Wheel
Engine
Door
Car
Note:
public void accept(CarElementVisitor visitor) {
visitor.visit(this);
}
Note:
public void accept(CarElementVisitor visitor) {
for(CarElement element : this.getElements()) {
element.accept(visitor);
}
visitor.visit(this);
}
«interface»
CarElementVisitor
+ visit(Wheel wheel)
+ visit(Engine engine)
+ visit(Body body)
+ visit(Car car)
CarElementCheckIn
variantVisitor
CarElementPrintVisitor

The specifications of the interfaces of the pattern

```
public interface CarElement {

    // CarElements have to provide accept().
    void accept(CarElementVisitor visitor);

// check if they are in a safe state
public boolean invariant ();
}

public interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Doors doors);
    void visit(Car car);
}
```

The class to be visited

```
public class Car implements CarElement{
    CarElement[] elements;

  public CarElement[] getElements() {
     return elements.clone(); // Return copy of array of references.
    }

    public Car() {
        this.elements = new CarElement[]
          { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Doors(), new Engine(8) };
    }

    public String toString(){
      return "\n  *** A Car *** \n";
    }
```

```
    public boolean invariant (){
    return (elements!=null && elements.length>0);
    }

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
        for(CarElement element : this.getElements()) {
            element.accept(visitor);
        }

    }
}
```

Now we need to code the components to be visited…

```
public class Doors implements CarElement {

private boolean leftDoorLocked;
private boolean rightDoorLocked;

public Doors(){
leftDoorLocked = true;
rightDoorLocked = true;
}

public boolean invariant(){return !(leftDoorLocked && !rightDoorLocked);}

public String toString(){
return "LeftDoorLocked is "+leftDoorLocked+" and RightDoorLocked is
"+rightDoorLocked;
}

public void lockLeft(){leftDoorLocked=true; rightDoorLocked=true;}
public void lockRight() {rightDoorLocked=true;}
public void unlockLeft(){leftDoorLocked=false; rightDoorLocked=false;}
public void unlockRight(){rightDoorLocked=false;}

public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```java
public class Engine implements CarElement {

private int speed;

private final int max;

public Engine(int max){
this.max=max;
speed=0;
}

public boolean invariant (){
return (max >0 && speed>=0 && speed<=max);
}

public String toString(){
return "Engine speed is "+speed+" / "+max;
}

    public void accelerate(){if (speed<max)speed++;}
    public void decelerate() {if (speed>0) speed--;}

    public void toMax(){speed=max;}

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
public class Wheel implements CarElement {
    private String name;

    private boolean turning;
    private int speed;

    public Wheel(String name) {
        this.name = name;
        turning = false;
        speed = 0;
    }

    public void accelerate(){speed++; turning = !(speed==0);}
    public void decelerate() {speed--;turning = !(speed==0);}

    public void stop(){speed=0; turning = false;}

    public boolean invariant(){return !(turning && speed==0);}

    public String toString() {
    if (!turning) return name + " is not turning";
    else return name + " is  turning with speed "+speed;
    }

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

Now the first visitor …

```
public class CarElementPrintVisitor implements CarElementVisitor {

    public void visit(Wheel wheel) {
        System.out.println(wheel);
    }

    public void visit(Engine engine) {
        System.out.println(engine);
    }

    public void visit(Doors doors) {
        System.out.println(doors);
    }

    public void visit(Car car) {
        System.out.println(car);
    }
}
```

Note: this is an artificial example – normally the visit code would be different for each component

Now the second visitor …

```
public class CarElementCheckInvariantVisitor implements CarElementVisitor {

    public void visit(Wheel wheel) {
        if (!wheel.invariant()){
        System.out.println(wheel +"in unsafe state, so it is stopped");
        wheel.stop();}       }
   }

    public void visit(Engine engine) {
    if (!engine.invariant()) {
    System.out.println(engine +",  this is an unsafe state, so it is set to max");
    engine.toMax();}
    }

    public void visit(Doors doors) {
    if (!doors.invariant()){
    System.out.println(doors +"in unsafe state,so both are locked");
        doors.lockLeft();}
    }

    public void visit(Car car) {
     if (!car.invariant()) System.out.println(car +"in unsafe state");
    }
}
```

This is more realistic than the first!

Now the test…

```
public class CarElementPrintVisitor implements CarElementVisitor {

public class Visitor_Test {

     public static void main(String[] args){
       Car car = new Car();
      car.accept(new CarElementPrintVisitor());
      car.accept(new CarElementCheckInvariantVisitor());
    }
}
```

TO DO: without executing the test program, try to guess what will be output to the screen when it runs.

TO DO: Download the visitor example from
`http://www-public.int-evry.fr/~gibson/Teaching/DesignPatterns/VisitorSource.zip`

Add a visitor which changes the state of each component of the car so that their invariants are broken

```
public class CarElementBreakInvariantVisitor implements
CarElementVisitor {

// TO DO
}
```

Update the test class to check that this visitor works correctly