
# Visitor

# 19

When you work with the Visitor, another behavioral pattern, you are always dealing with a collection of objects with different interfaces. Related operations are to be performed on all objects, which are grouped together in their own class, the Visitor.

## 19.1 A Simple Example

A car is an aggregate; it consists of wheels, an engine and the chassis. The car as well as its components are grouped under the interface Element.

### 19.1.1 The Power Unit

Please analyze the classes in the package `visitordemo.car` of the sample project Visitor. You will find that each element has its own data fields and methods. Besides, the methods `getDescription()` and `accept()` are prescribed in the interface. One method will return a short description, the other will provide an interface for a Visitor object. Take the `Wheel` class as an example. The wheel can store its position – for example, front left. It can also store and return the air pressure. The `accept()` method is overridden so that the `visit()` method is called on the Visitor object passed in, and passing over its own instance – `this`.

```
public class Wheel implements Element {
    private final Position position;
    private final double pressure = 2.0;
```



O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_19

```
        public double getPressure() {
            return pressure;
        }

        public wheel(Position position) {
            this.position = position;
        }

        @Override
        public void accept(VisitorIF visitor) {
            visitor.visit(this);
        }

        @Override
        public String getDescription() {
            return "- Wheel " + position;
        }
}
```

On the chassis, the wheels are mounted. When the accept method is called on it, in the first step it calls the visit method on the Visitor and passes itself as a parameter. In the second step, the visit method is called for each wheel.

```
public class Chassis implements Element {
        private final wheel[] wheels;

        public Rad[] getWheels() {
            return wheels;
        }

        Chassis(wheel[] wheels) {
            this.wheels = wheels;
        }

        @Override
        public void accept(VisitorIF visitor) {
            visitor.visit(this);
            for (var wheel : wheels)
                wheel.accept(visitor);
        }

        @Override
        public String getDescription() {
            return "- Chassis";
        }
}
```

The car should also still be described in the required brevity. It stores the information whether the tank is full, whether there is enough oil and whether blinker water (this innovative concoction is here representative for any other possible consumption liquid) was refilled. There are corresponding access methods to these attributes. The constructor installs a new engine, creates the chassis and attaches the wheels to it.

```
public class Car implements Element {
       private final List<Element> parts =
                                          new ArrayList<>();

       // … abridged


       public Car() {
           parts.add(new Engine());
           parts.add(new Chassis(new Wheel[] {
               new Wheel(Position.FL),
               new Wheel(Position.FR),
               new Wheel(Position.RL),
               new Wheel(Position.RR)
           }));
       }
}
```

The `accept()` method calls each component's visit method and passes them a reference to the visitor. Then the visit method is called on the Visitor and the car passes itself over as a parameter. I won't print the Engine class here – it follows the same logic as the previous classes.

### 19.1.2 The Visitor

I define the Visitor class in a separate package, `visitordemo.visitor`. A visitor is described by the interface `VisitorIF`. Every class that should be able to act as a visitor must override the methods declared in it; this concerns the method `visit()`, which is overloaded four times, once for each element.

The Visitor inheritance hierarchy is tightly bound to the Element classes. It must have precise knowledge of the interfaces of the objects to be visited, as you will see in a moment. Conversely, the element classes only need to know the Visitor interface.

```
public interface VisitorIF {
void visit(wheel wheel);
void visit(Engine engine);
void visit(Chassis chassis);
void visit(Car car);
}
```

A concrete Visitor is the `PartsVisitor`. It compiles a list of the descriptions of all parts. To do this, it overwrites the method `visit()` so that the method `getDescription()` is called on the parameter passed, i.e. a wheel, an engine, a chassis or a car. The descriptions of all the individual parts are combined by a StringBuilder. The method `getParts()` returns a string with the list of all parts.

```
public class PartsVisitor implements VisitorIF {
       private final StringBuilder builder =
                                        new StringBuilder();
       public String getPartsList() {
           return "Components: \n" + builder.toString();
       }

       @Override
       public void visit(Wheel wheel) {
           builder.append(wheel.getDescription()).
                                           append("\n");
       }

       @Override
       public void visit(Engine engine) {
           builder.append(engine.getDescription()).
                                           append("\n");
       }

       // … abridged


}
```

How can the client now use this construction?

### 19.1.3 The Client

The client creates an instance of the Auto class and a component Visitor. It then calls the `accept()` method with the Visitor as a parameter. The resulting string is output to the console.

```
public class ApplStart {
       public static void main(String[] args) {
           var car = new Car();
           var visitor = new PartsVisitor();
           car.accept(visitor);
           var parts = visitor.getPartsList();
           System.out.println(parts);

           // … abridged
       }
}
```

The list of components compiled by the Visitor is displayed on the console:

```
 Components:
- motor
- chassis
- Wheel front left
- Wheel front right
- Wheel rear left
- Wheel rear right
- Car (remaining components)
```

### 19.1.4 Another Visitor

When you drive to the workshop because of an irregularity on your vehicle, a person there hooks up a diagnostic device to a defined interface in the car and can shortly afterwards say precisely whether the car has a defect, and if so, in which element. This situation is mapped by the DiagnoseVisitor. It works very similarly to the Component Visitor. It queries certain information on each element and evaluates it. There are two different procedures. The car can measure by itself if the tank is full and if the oil level is correct; therefore, it returns a boolean value with the appropriate access methods. I assume that very few cars measure the air pressure in their tires independently, although this is technically possible.

Here the visitor gets the current value and evaluates it. At the end it can give information whether the car is ready to drive. Please analyze the DiagnoseVisitor independently – the class is not complicated.

### 19.1.5 Criticism of the Project

It becomes clear what I meant at the beginning by related operations: There are a number of classes on each of which a diagnosis is to be performed. Instead of defining the diagnostic methods in the respective classes, they are combined in a separate class. This has two advantages: First, the classes are not "polluted" by code that is not part of their actual core business. In addition, the newly defined operations can be maintained much more easily; they are located in a single class and do not have to be maintained separately in umpteen classes.

The classes involved do not have to have a common interface. In the "Visitor" project, both the car and the components were subclasses of the Element interface. However, this is not mandatory in the sense of the Visitor pattern.

A disadvantage of the Visitor Pattern is that the visited object must have a sufficiently extensive interface to be able to query all relevant data. For example, the wheel class must be able to provide information about air pressure, the engine must be able to name its state, and so on. You may even need to use the Visitor Pattern to provide access to data that you intended to encapsulate.

Another disadvantage is that you will have extensive maintenance on all concrete Visitor classes when a new element is inserted. Imagine that you additionally define the class Brake – the interface Visitor must declare a corresponding visit method, which must be overridden by all concrete Visitor classes. The Visitor pattern is always best used when new operations are needed rather than changing the object structure more frequently.

> The Visitor Pattern is an excellent example of the Open-Closed Principle. You have a set of classes that are closed against change. But they leave the door ajar for extensions by providing an interface that allows other objects to define new behavior.

One last aspect should be addressed: the type of iteration. In the project you have just worked on, you will find an internal driver – you call the `accept()` method on the Car object, which ensures that iteration is performed over all components. You could also conceivably define an external iterator. The solution with an internal iterator does not have to be the one you find practical.

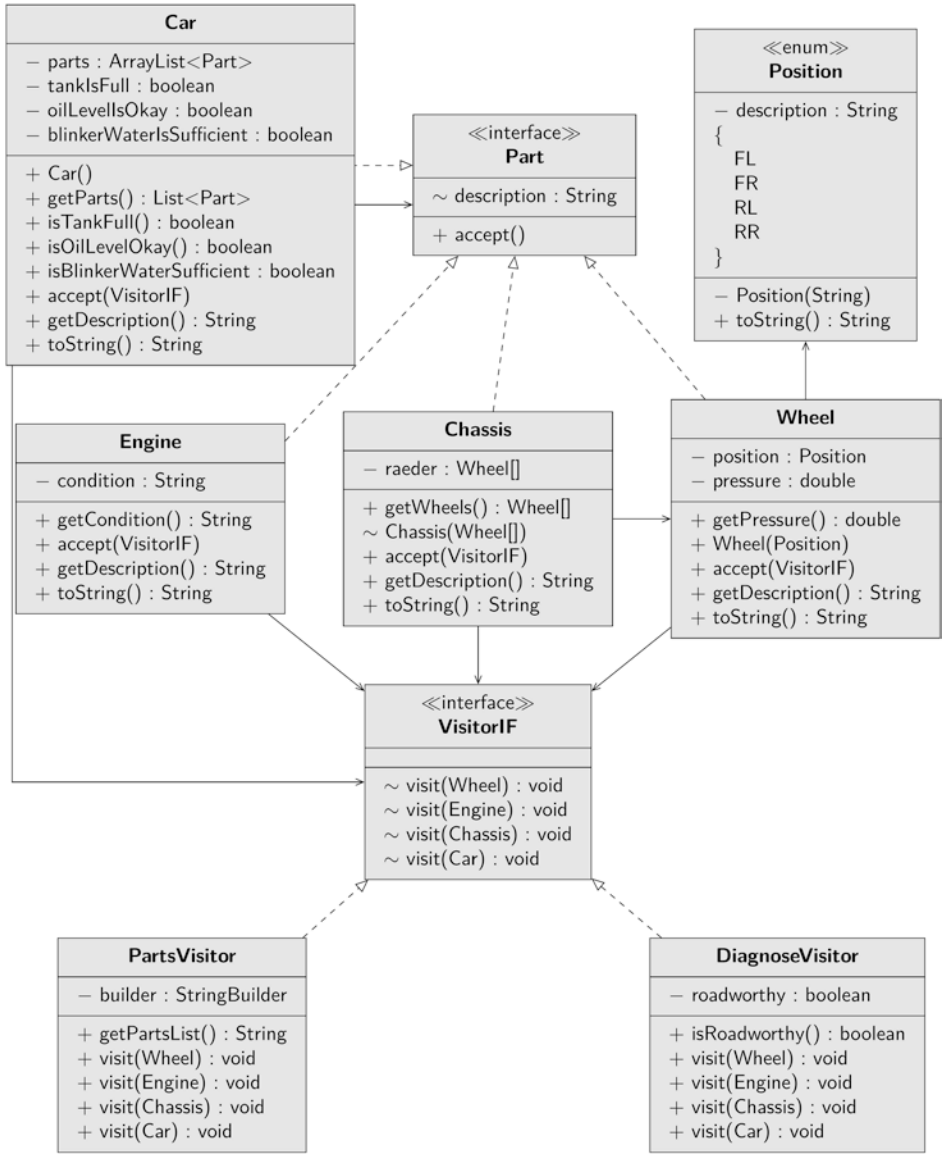


**Fig. 19.1** UML diagram of the Visitor pattern (example project Visitor)

## 19.2 Visitor – The UML Diagram

You can see the UML diagram from the Visitor sample project in Fig. 19.1.

## 19.3 Summary

Go through the chapter again in key words:

- You are dealing with objects with different interfaces.
- Related operations – a diagnosis – should be performed on all objects.
- The operations are combined in a single class, the Visitor.
- Each object provides an interface that the Visitor can call.
- The object calls the visitor's overloaded visit method on the visitor and passes itself as a parameter.
- The Visitor defines a visit method for each object in the collection.
- According to the rules of polymorphism, the appropriate visit method for the object is called.

## 19.4 Description of Purpose

The Gang of Four describes the purpose of the "Visitor" pattern as follows:

> Encapsulate an operation to be performed on the elements of an object structure as an object. The visitor pattern allows you to define a new operation without changing the classes of the elements it operates on.