

COMP2511

Visitor Pattern

Prepared by

Dr. Ashesh Mahidadia

Design Patterns

Creational Patterns

- ❖ Factory Method
- ❖ Abstract Factory
- ❖ Builder
- ❖ Singleton

Structural Patterns

- ❖ Adapter
- ❖ Composite
- ❖ Decorator

Behavioral Patterns

- ❖ Iterator
- ❖ Observer
- ❖ State
- ❖ Strategy
- ❖ Template
- ❖ Visitor

The lecture slides use material from the websites
<https://refactoring.guru/design-patterns/>
and the wikipedia pages.

Visitor Pattern

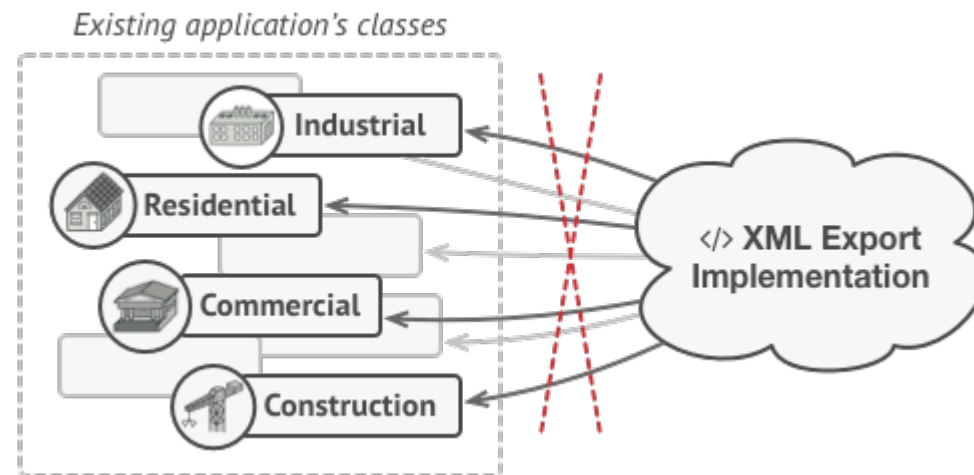
Visitor Pattern

- ❖ Visitor is a **behavioral** design pattern that adds new operations/behaviors to the existing objects, without modifying them.
- ❖ The visitor design pattern is a way of **separating** an algorithm from an object structure on which it operates.
- ❖ A practical result of this separation is the ability to **add new operations** to existing object structures without modifying the structures.
- ❖ It is one way to follow the **open/closed principle**.
- ❖ A **visitor class** is created that **implements** all of the appropriate specializations of the **virtual operation/method**.
- ❖ The visitor **takes** the **instance reference** as input, and implements the goal (additional behavior).
- ❖ Visitor pattern can be added to **public APIs**, allowing its clients to perform operations on a class without having to modify the source.

Visitor Pattern

Problem:

- A geographic information **structured** as one colossal **graph**.
- Each **node** of the graph may represent **a city, an industry, a sightseeing area, etc.**
- Each node type is represented by its own class, while each specific node is an object.
- **Task:** you want to **export** the graph into **XML format**.



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

Visitor Pattern

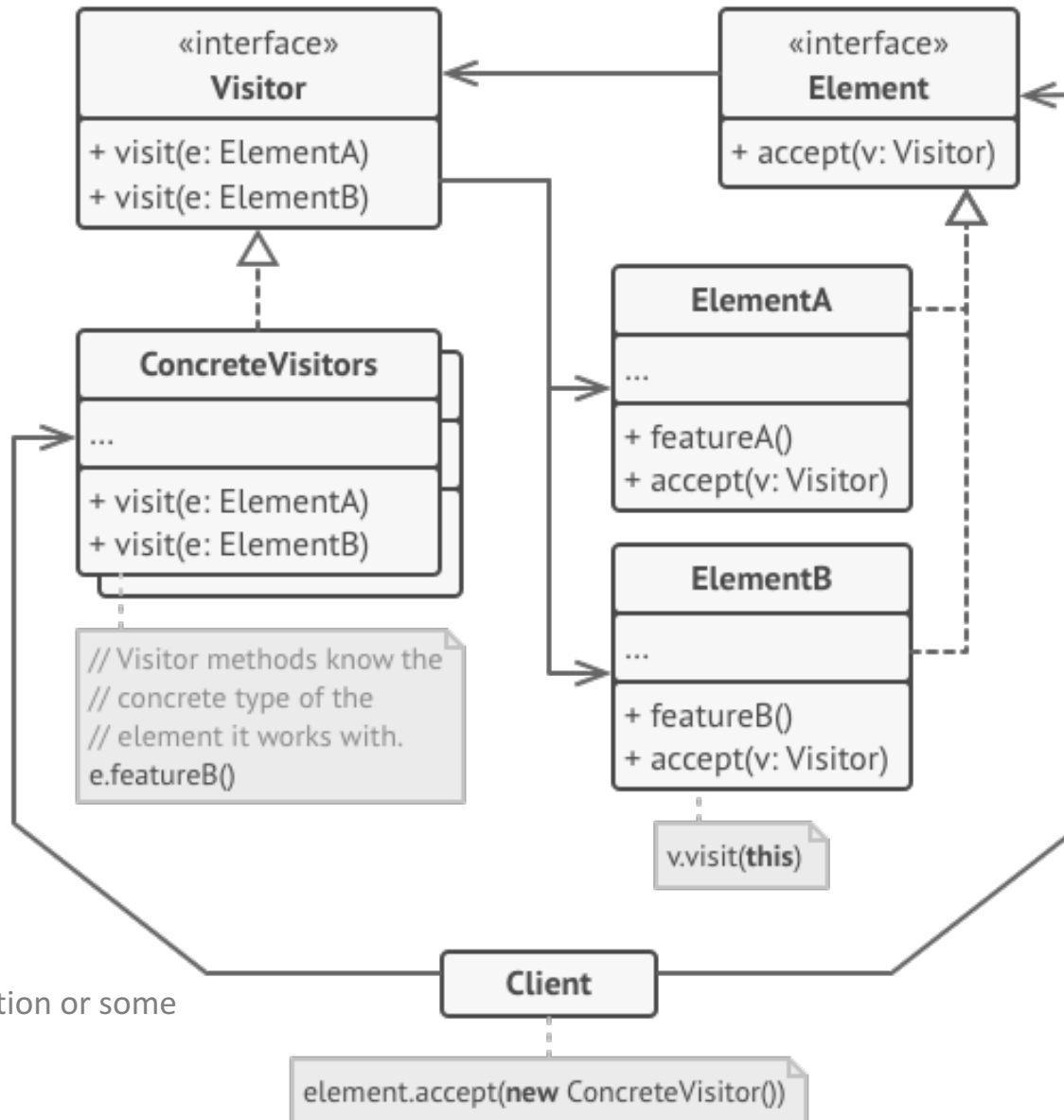
Solution:

- ❖ The Visitor pattern suggests that you place the new behavior into a **separate class** called **visitor**, instead of trying to integrate it into existing classes.
- ❖ The original object that had to perform the behavior is now passed to one of the **visitor's methods as an argument**, providing the method access to all necessary data contained within the object (see the example for more clarification).
- ❖ The **visitor class** need to define **a set of methods**, one for each type. For example, a city, a sightseeing place, an industry, etc.
- ❖ The visitor pattern uses a technique called “**Double Dispatch**” to execute a suitable method on a given object (of different types).
 - An object “**accepts**” a **visitor** and tells it what visiting method should be executed. See the example for more clarifications.
 - One additional method allows us to **add further behaviors without** further altering the code.

Visitor Pattern: Structure

1 The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments.

2 Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.



3 The **Element** interface declares a method for “accepting” visitors. This method should have one parameter declared with the type of the visitor interface.

4 Each **Concrete Element** must implement the acceptance method. The purpose of this method is to **redirect the call to the proper visitor’s method corresponding to the current element class**. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.

5 The **Client** usually represents a collection or some other complex object (for example, a Composite tree).

Visitor Pattern: Example-1

Read the **example code** discussed in the lectures, and also **provided** for this week

```
public interface CarElement {  
    void accept(CarElementVisitor visitor);  
}
```

```
public class Wheel implements CarElement {  
  
    private final String name;  
    private double cost = 210;  
  
    public Wheel(final String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public void accept(CarElementVisitor visitor) {  
        visitor.visit(this);  
    }  
}
```

“accept” method for Simple Object

```
package car;  
  
import java.util.ArrayList;  
  
public class Car implements CarElement {  
    private ArrayList<CarElement> elements =  
        new ArrayList<CarElement>();  
  
    private double cost = 6000;  
  
    public Car() {  
        this.elements.add( new Wheels() );  
        this.elements.add( new Body() );  
        this.elements.add( new Engine() );  
    }  
  
    @Override  
    public void accept(CarElementVisitor visitor) {  
        for (CarElement element : elements) {  
            element.accept(visitor);  
        }  
        visitor.visit(this);  
    }  
}
```

“accept” method for Composite Object

Visitor Pattern: Example-1

```
public interface CarElementVisitor {  
  
    void visit(Body body);  
    void visit(BodyPart1 bodyPart1);  
    void visit(BodyPart2 bodyPart2);  
  
    void visit(Wheels wheels);  
    void visit(Wheel wheel);  
  
    void visit(Car car);  
    void visit(Engine engine);  
  
}
```

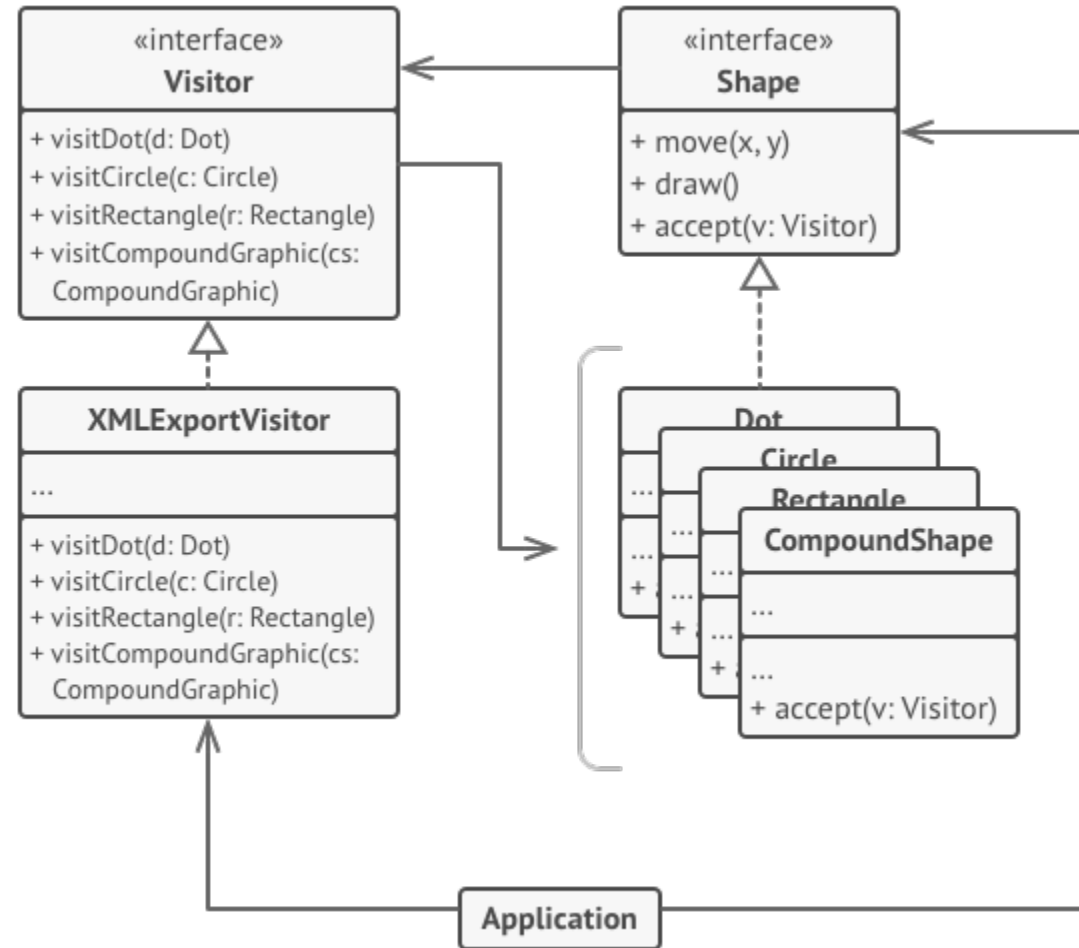
The visitor class need to define a set of visit methods, each of which could take arguments of different types. For example, "Body", "Engine", "Car", "Wheel", etc. Need to implement visit methods for each type.

```
public class CarElementPrintVisitor implements CarElementVisitor {  
  
    @Override  
    public void visit(Body body) {  
        System.out.println("Visiting body");  
    }  
  
    @Override  
    public void visit(Car car) {  
        System.out.println("Visiting car");  
    }  
  
    @Override  
    public void visit(Engine engine) {  
        System.out.println("Visiting engine");  
    }  
  
    @Override  
    public void visit(Wheel wheel) {  
        System.out.println("Visiting " + wheel.getName() + " wheel");  
    }  
  
    @Override  
    public void visit(BodyPart1 bodyPart1) {  
        System.out.println("Visiting bodyPart1");  
    }  
  
}
```

```
public static void main(final String[] args) {  
  
    Car car = new Car();  
  
    System.out.println("\n ----- From CarElement  
    car.accept(new CarElementPrintVisitor());  
}
```

Read the example code discussed in the lectures, and also provided for this week

Visitor Pattern: Example-2



Example in Java (MUST read):

<https://refactoring.guru/design-patterns/visitor/java/example>

Visitor Pattern: Applicability and Limitation

Applicability:

Moving operations into visitor classes is beneficial when,

- ❖ many **unrelated operations** on an object structure are required,
- ❖ the classes that make up the object structure are known and **not expected to change**,
- ❖ **new** operations need to be added **frequently**,
- ❖ an **algorithm** involves several classes of the object structure, but it is desired to manage it **in one single location**,
- ❖ an **algorithm** needs to work **across** several independent class hierarchies.

Limitation:

- ❖ extensions to the class hierarchy more difficult,
as a **new class** typically **require a new visit method** to be added to each visitor.

Visitor Pattern

- ❖ More information on “Visitor and Double Dispatch”:
<https://refactoring.guru/design-patterns/visitor-double-dispatch>
- ❖ For more, read the following:
<https://refactoring.guru/design-patterns/visitor>

End