Visitor Pattern

Resources

- 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- 36. Visitor Design Pattern | Double Dispatch | Low Level Design

Definition

> **The Visitor design pattern is a behavioral design pattern that allows you to add new operations to existing object structures without modifying those structures. It achieves this by separating the algorithm from the actual objects that it operates on. It uses Double Dispatch to implement this.**

Double Dispatch is a method where the runtime type of the caller object (the receiver) and the runtime type of the argument object (the visitor) determine which method is invoked.

In simple terms, Double Dispatch is the selection of a method based on the runtime types of two objects, the receiver and the argument.

More on this later...

The Problem: Naive Approach

```java
// Bloated Element class with multiple operations
public class SuiteHotelRoom {
    private String roomNumber;
    private String numberOfRooms;

    public SuiteHotelRoom(String roomNumber, String numberOfRooms) {
        this.roomNumber = roomNumber;
        this.numberOfRooms = numberOfRooms;
    }

    public void clean() {
        System.out.println("Housekeeping: Cleaning suite " +
                roomNumber + " with " +
                numberOfRooms + " rooms (90 minutes)");
    }

    public void deliverRoomService(String orderDetails) {
        System.out.println("Room Service: VIP delivery of " +
orderDetails +
```

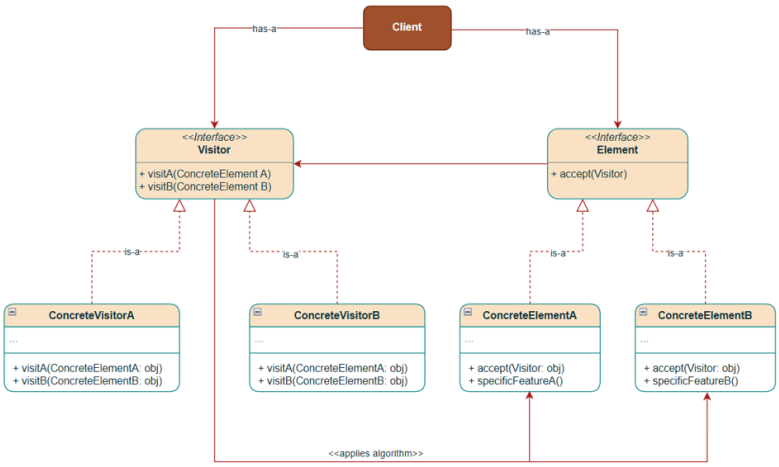
```
                " to suite " + roomNumber +
                " with full dining setup");
    }

    public double calculatePrice() {
        System.out.println("Pricing: Suite " + roomNumber +
                " - Rs. 2000/night");
        return 500.0;
    }

    // many more operations can come over time
}
```

```
// Usage
public class Demo {
    public static void main(String[] args) {
        System.out.println("##### Visitor Pattern: Problem Demo
#####");
        SuiteHotelRoom suite = new SuiteHotelRoom("301", "2");
        suite.clean();
        suite.deliverRoomService("Breakfast");
        suite.calculatePrice();
    }
}
```

The Key Problems Without Visitor Pattern

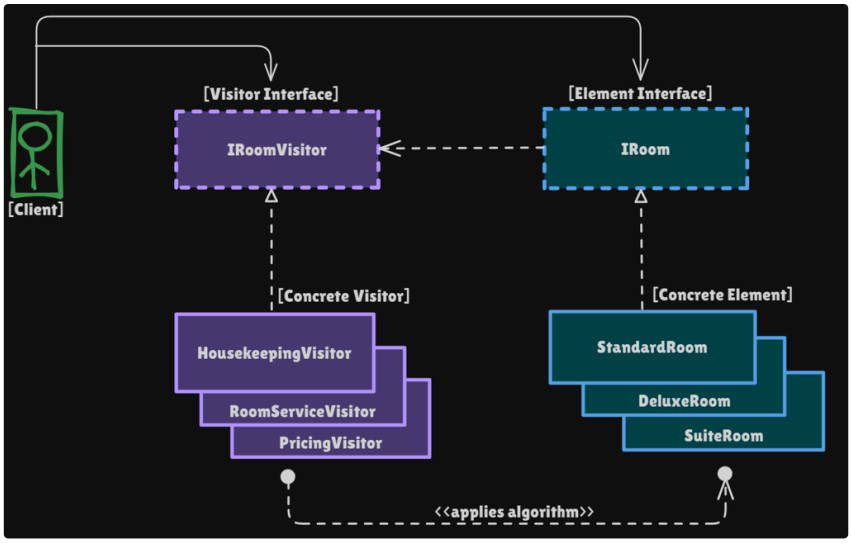
1. Violates the Open/Closed Principle and the Single Responsibility Principle, making it unextensible.
2. Scattered Operation Logic leading to redundant code.
3. Difficult Testing
4. Tight coupling & Poor Reusability of code

Class Diagram



Structure of Visitor Pattern

Understanding the Structure of the Visitor Pattern using the Hotel Room example:



- **Element Interface(** `IRoom` **)**: Declares a contract for all concrete element types to implement the `accept(IRoomVisitor visitor)` method, which allows visitors to perform operations on them.
- **Concrete Element(** `StandardRoom` , `DeluxeRoom` , and `SuiteRoom` **)**: These concrete element types are actual objects on which the algorithm is applied. Inside the `accept(IRoomVisitor visitor)` method, they call the appropriate visitor method for their respective types.
- **Visitor Interface(** `IRoomVisitor` **)**: Defines one `visit(ConcreteElement e)` method for each room type. This is the core logic of the pattern. The algorithm is executed on the object.
- **Concrete Visitor(** `HousekeepingVisitor` , `RoomServiceVisitor` , `PricingVisitor` **)**: Each visitor service implements how to handle the particular operation on each element type differently.

Implementation

```java
// Visitor interface - defines operations
public interface IRoomVisitor {
    void visitStandardRoom(StandardRoom room);

    void visitDeluxeRoom(DeluxeRoom room);

    void visitSuiteRoom(SuiteRoom room);
}
```

```java
// Element interface - represents rooms(elements) that can be visited
public interface IRoom {
    void accept(IRoomVisitor visitor);

}
```

```java
// Concrete Element - Different room types
public class StandardRoom implements IRoom {
    private final String roomNumber;

    public StandardRoom(String roomNumber) {
        this.roomNumber = roomNumber;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitStandardRoom(this);
    }

    public String getRoomNumber() {
```

```
        return roomNumber;
    }
}
```

```
// Concrete Element - Different room types
public class DeluxeRoom implements IRoom {
    private final String roomNumber;
    private final boolean hasJacuzzi;

    public DeluxeRoom(String roomNumber, boolean hasJacuzzi) {
        this.roomNumber = roomNumber;
        this.hasJacuzzi = hasJacuzzi;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitDeluxeRoom(this);
    }

    public String getRoomNumber() {
        return roomNumber;
    }

    public boolean hasJacuzzi() {
        return hasJacuzzi;
    }
}
```

```
// Concrete Element - Different room types
public class SuiteRoom implements IRoom {
    private final String roomNumber;
    private final int numberOfRooms;

    public SuiteRoom(String roomNumber, int numberOfRooms) {
        this.roomNumber = roomNumber;
        this.numberOfRooms = numberOfRooms;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitSuiteRoom(this);
    }

    public String getRoomNumber() {
        return roomNumber;
    }

    public int getNumberOfRooms() {
        return numberOfRooms;
    }
}
```

```
// Concrete Element - Different room types
public class SuiteRoom implements IRoom {
    private final String roomNumber;
    private final int numberOfRooms;

    public SuiteRoom(String roomNumber, int numberOfRooms) {
        this.roomNumber = roomNumber;
        this.numberOfRooms = numberOfRooms;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitSuiteRoom(this);
    }

    public String getRoomNumber() {
        return roomNumber;
    }

```

```
    public int getNumberOfRooms() {
        return numberOfRooms;
    }
}
```

```
// Concrete Element - Different room types
public class SuiteRoom implements IRoom {
    private final String roomNumber;
    private final int numberOfRooms;

    public SuiteRoom(String roomNumber, int numberOfRooms) {
        this.roomNumber = roomNumber;
        this.numberOfRooms = numberOfRooms;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitSuiteRoom(this);
    }

    public String getRoomNumber() {
        return roomNumber;
    }

    public int getNumberOfRooms() {
        return numberOfRooms;
    }
}
```

```
// Concrete Element - Different room types
public class SuiteRoom implements IRoom {
    private final String roomNumber;
    private final int numberOfRooms;

    public SuiteRoom(String roomNumber, int numberOfRooms) {
        this.roomNumber = roomNumber;
        this.numberOfRooms = numberOfRooms;
    }

    @Override
    public void accept(IRoomVisitor visitor) {
        visitor.visitSuiteRoom(this);
    }

    public String getRoomNumber() {
        return roomNumber;
    }

    public int getNumberOfRooms() {
        return numberOfRooms;
    }
}
```

```
// Concrete Visitor - demonstrates adding new operations easily
public class HousekeepingVisitor implements IRoomVisitor {
    @Override
    public void visitStandardRoom(StandardRoom room) {
        System.out.println("Housekeeping: Cleaning standard room " +
                room.getRoomNumber() + " (30 minutes)");
    }

    @Override
    public void visitDeluxeRoom(DeluxeRoom room) {
        System.out.println("Housekeeping: Cleaning deluxe room " +
                room.getRoomNumber() +
                (room.hasJacuzzi() ? " including jacuzzi" : "") +
                " (45 minutes)");
    }

    @Override
    public void visitSuiteRoom(SuiteRoom room) {
```

```
        System.out.println("Housekeeping: Cleaning suite " +
                room.getRoomNumber() + " with " +
                room.getNumberOfRooms() + " rooms (90 minutes)");
    }
}
```

```
// Pricing visitor - demonstrates adding new operations easily
public class RoomServiceVisitor implements IRoomVisitor {
    private final String orderDetails;

    public RoomServiceVisitor(String orderDetails) {
        this.orderDetails = orderDetails;
    }

    @Override
    public void visitStandardRoom(StandardRoom room) {
        System.out.println("Room Service: Delivering " + orderDetails
+
                " to standard room " + room.getRoomNumber());
    }

    @Override
    public void visitDeluxeRoom(DeluxeRoom room) {
        System.out.println("Room Service: Premium delivery of " +
orderDetails +
                " to deluxe room " + room.getRoomNumber() +
                " with complimentary champagne");
    }

    @Override
    public void visitSuiteRoom(SuiteRoom room) {
        System.out.println("Room Service: VIP delivery of " +
orderDetails +
                " to suite " + room.getRoomNumber() +
                " with full dining setup");
    }
}
```

```
// Pricing visitor - demonstrates adding new operations easily
public class PricingVisitor implements IRoomVisitor {
    private double totalRevenue = 0;

    @Override
    public void visitStandardRoom(StandardRoom room) {
        double price = 1000.0;
        totalRevenue += price;
        System.out.println("Pricing: Standard room " +
room.getRoomNumber() +
                " - Rs." + price + "/night");
    }

    @Override
    public void visitDeluxeRoom(DeluxeRoom room) {
        double price = 2000.0;
        totalRevenue += price;
        System.out.println("Pricing: Deluxe room " +
room.getRoomNumber() +
                " - Rs." + price + "/night");
    }

    @Override
    public void visitSuiteRoom(SuiteRoom room) {
        double price = 5000.0;
        totalRevenue += price;
        System.out.println("Pricing: Suite " + room.getRoomNumber() +
                " - Rs." + price + "/night");
    }

    public double getTotalRevenue() {
        return totalRevenue;
    }
```

```
}
```

```
// Usage
public class HotelVisitorDemo {
    public static void main(String[] args) {
        System.out.println("\n###### Visitor Design Pattern Demo ######");

        // Create different room types(elements) - Standard, Deluxe, Suite
        IRoom[] rooms = {
                new StandardRoom("101"),
                new DeluxeRoom("201", true),
                new SuiteRoom("301", 3),
                new StandardRoom("102"),
                new DeluxeRoom("202", false)
        };

        // Calling Visitors on elements
        System.out.println("\n==> Housekeeping Service");
        IRoomVisitor housekeeping = new HousekeepingVisitor();
        for (IRoom room : rooms) {
            room.accept(housekeeping);
        }

        System.out.println("\n==> Room Service");
        IRoomVisitor roomService = new RoomServiceVisitor("Breakfast");
        rooms[0].accept(roomService);  // Deliver to standard room
        rooms[1].accept(roomService);  // Deliver to deluxe room
        rooms[2].accept(roomService);  // Deliver to suite

        System.out.println("\n==> Revenue Calculation");
        PricingVisitor pricing = new PricingVisitor();
        for (IRoom room : rooms) {
            room.accept(pricing);
        }
        System.out.println("Total Revenue: Rs." + pricing.getTotalRevenue());
    }
}
```

Output

```
###### Visitor Design Pattern Demo ######

==> Housekeeping Service
Housekeeping: Cleaning standard room 101 (30 minutes)
Housekeeping: Cleaning deluxe room 201 including jacuzzi (45 minutes)
Housekeeping: Cleaning suite 301 with 3 rooms (90 minutes)
Housekeeping: Cleaning standard room 102 (30 minutes)
Housekeeping: Cleaning deluxe room 202 (45 minutes)

==> Room Service
Room Service: Delivering Breakfast to standard room 101
Room Service: Premium delivery of Breakfast to deluxe room 201 with complimentary champagne
Room Service: VIP delivery of Breakfast to suite 301 with full dining setup

==> Revenue Calculation
Pricing: Standard room 101 - Rs.1000.0/night
Pricing: Deluxe room 201 - Rs.2000.0/night
Pricing: Suite 301 - Rs.5000.0/night
Pricing: Standard room 102 - Rs.1000.0/night
Pricing: Deluxe room 202 - Rs.2000.0/night
Total Revenue: Rs.11000.0

Process finished with exit code 0
```

How does the Visitor Pattern achieve Double Dispatch?

Single Dispatch

Single dispatch means the method that gets executed is determined by the runtime type of only one object, typically the receiver of the method call.

```
// No visitors

// Just different types of rooms - IRoom - Standard, Deluxe, Suite
// each implementing its own version of accept

IRoom roomObj = new DeluxeRoom();
roomObj.accept();

// At runtime we execute accept() method implementation of DeluxeRoom class
// The method executed depends  only on the runtime type of roomObj
// (which is DeluxeRoom), not on any other factors.
```

Double Dispatch

Double Dispatch comes into the picture when you need behavior, i.e., actual method execution, to depend on multiple object types:

1. The runtime type of the caller object (i.e., the receiver or element) → first dispatch
2. The runtime type of the object passed as an argument (i.e., the visitor) → second dispatch

```
// Different types of Visitors - IRoomVisitor - HousekeepingVisitor,
// RoomServiceVisitor and PricingVisitor

// Different types of rooms - IRoom - Standard, Deluxe, Suite
IRoom myroom = new StandardRoom("101");

IRoomVisitor myvisitor = new HousekeepingVisitor();

myroom.accept(myvisitor); // DOUBLE DISPATCH

// The runtime type of the caller object - myroom is resolved at runtime &
// StandardRoom.accept() is called ---> FIRST DISPATCH

// The runtime type of the object passed as an argument - inside the accept() method
// myvisitor is resolved at runtime & HousekeepingVisitor.visitStandardRoom()
// is called ---> SECOND DISPATCH
```

Visitor vs Strategy Design Pattern

Strategy Pattern: Encapsulates **alternative algorithms** for a single operation.

- Allows swapping different behaviors at runtime.
- These algorithms are independent of the object they are applied.
- Here, we find Different ways to do ONE thing.

Visitor Pattern: Adds **new operations** to an object structure without modifying it.

- Allows adding operations across multiple types.
- The new operations are tied to specific objects they are applied.
- Here we perform Different operations on MANY types.