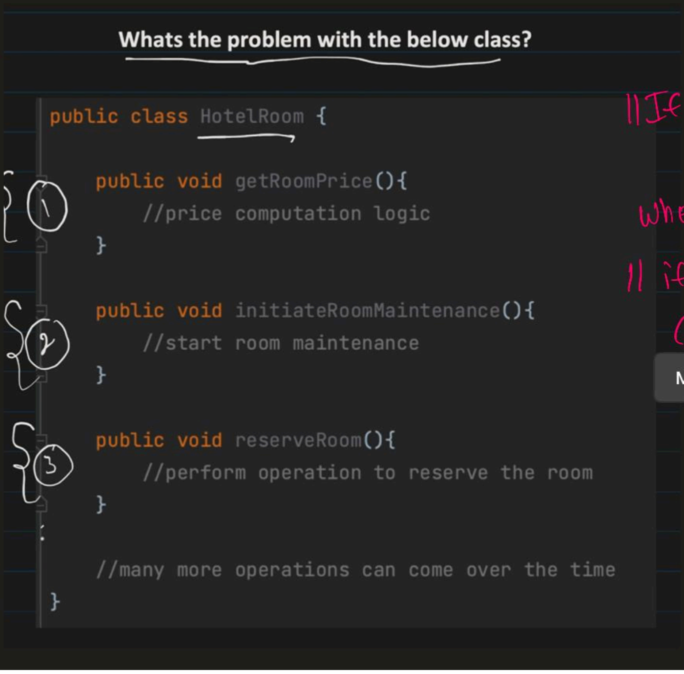
33

Hotel Booking System (Behavioural DP)

(Visiter Design Pattern)



Whats the problem with the below class?

```
public class HotelRoom {

    public void getRoomPrice(){
        //price computation logic
    }

    public void initiateRoomMaintenance(){
        //start room maintenance
    }

    public void reserveRoom(){
        //perform operation to reserve the room
    }

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

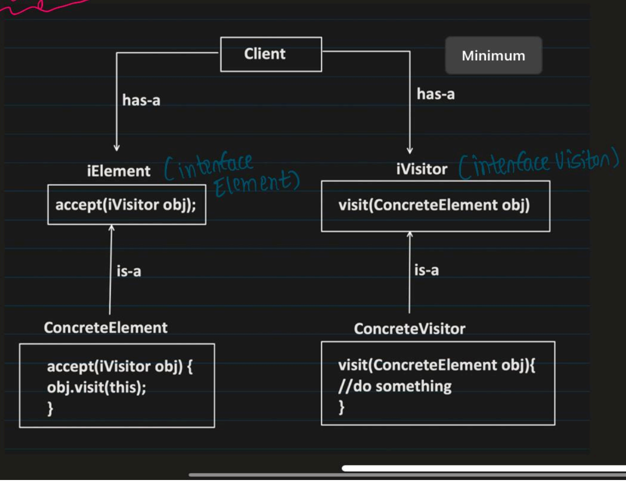
// If any new operation, added then test class again, when new operation is added.

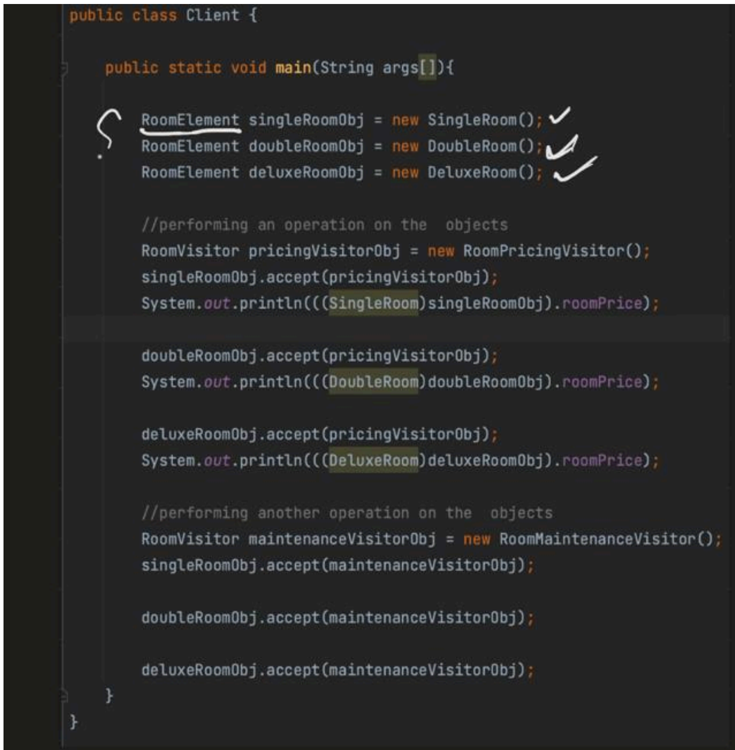
// if so methods, then class can grow huge.

Minimum

- It is a **Behavioral design pattern**
- That allows you to add new operations to existing classes without changing their structure.
- It achieves this by separating the ~~algorithm~~ operation from the objects on which it operates.
- It does **Double Dispatch** to achieve this.
  (Double Dispatch means, method which need to be invoked decided by the caller object and the object passed in the argument.)

UML Diagram



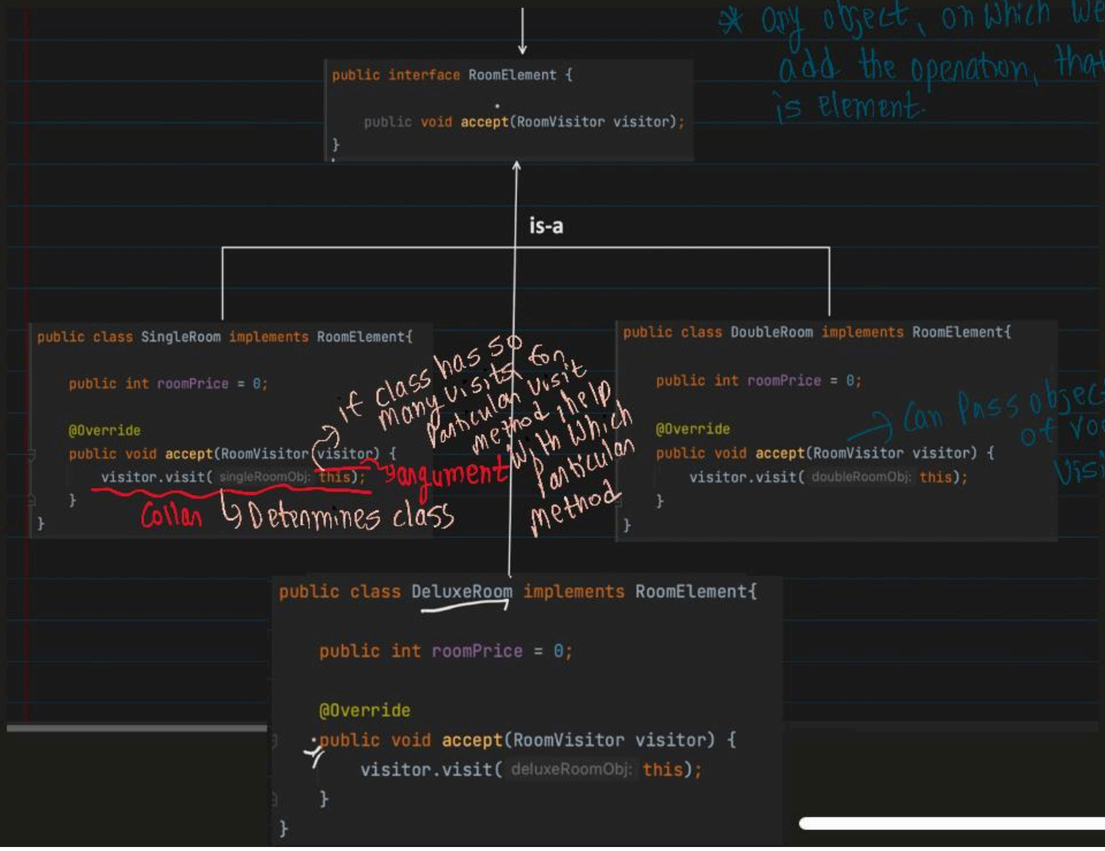


```java
public class Client {

    public static void main(String args[]){

        RoomElement singleRoomObj = new SingleRoom(); ✓
        RoomElement doubleRoomObj = new DoubleRoom(); ✓
        RoomElement deluxeRoomObj = new DeluxeRoom(); ✓

        //performing an operation on the  objects
        RoomVisitor pricingVisitorObj = new RoomPricingVisitor();
        singleRoomObj.accept(pricingVisitorObj);
        System.out.println(((SingleRoom)singleRoomObj).roomPrice);

        doubleRoomObj.accept(pricingVisitorObj);
        System.out.println(((DoubleRoom)doubleRoomObj).roomPrice);

        deluxeRoomObj.accept(pricingVisitorObj);
        System.out.println(((DeluxeRoom)deluxeRoomObj).roomPrice);

        //performing another operation on the  objects
        RoomVisitor maintenanceVisitorObj = new RoomMaintenanceVisitor();
        singleRoomObj.accept(maintenanceVisitorObj);

        doubleRoomObj.accept(maintenanceVisitorObj);

        deluxeRoomObj.accept(maintenanceVisitorObj);
    }
}
```

* earlier, createRoomClass, Inside that reserve room, Calculate Price, Do Maintanence. if More Methods Comes, add Visitor

Minimum

* any object, on which we have to add the operation, that object is element.



```java
public interface RoomElement {

    public void accept(RoomVisitor visitor);
}
```

```java
public class SingleRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( singleRoomObj: this);
    }
}
```

if class has so many visits for Particular visit method the help with which Particular method

argument

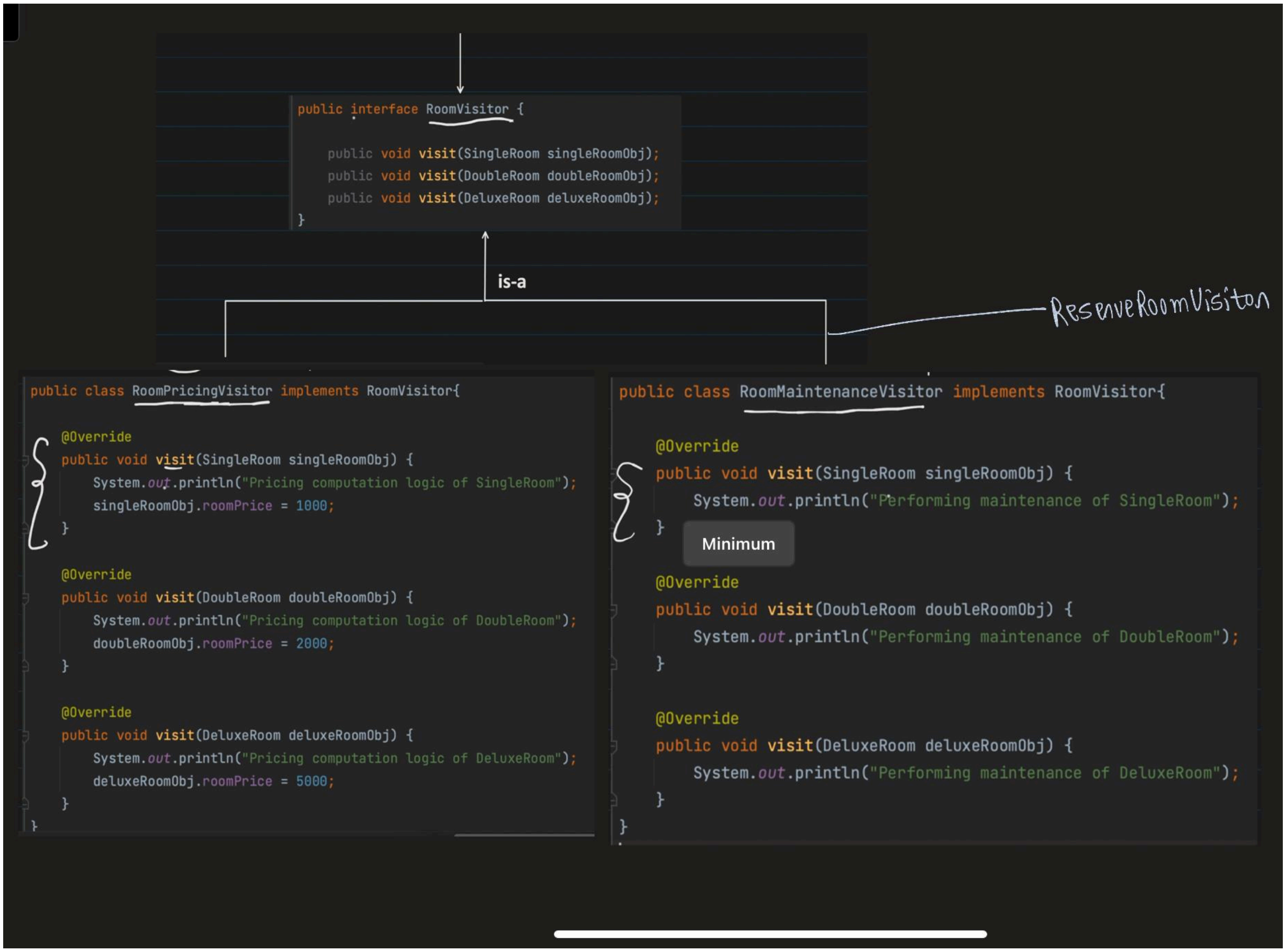
Caller

↳ Determines class

```java
public class DoubleRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( doubleRoomObj: this);
    }
}
```

→ Can Pass object of room Visitor

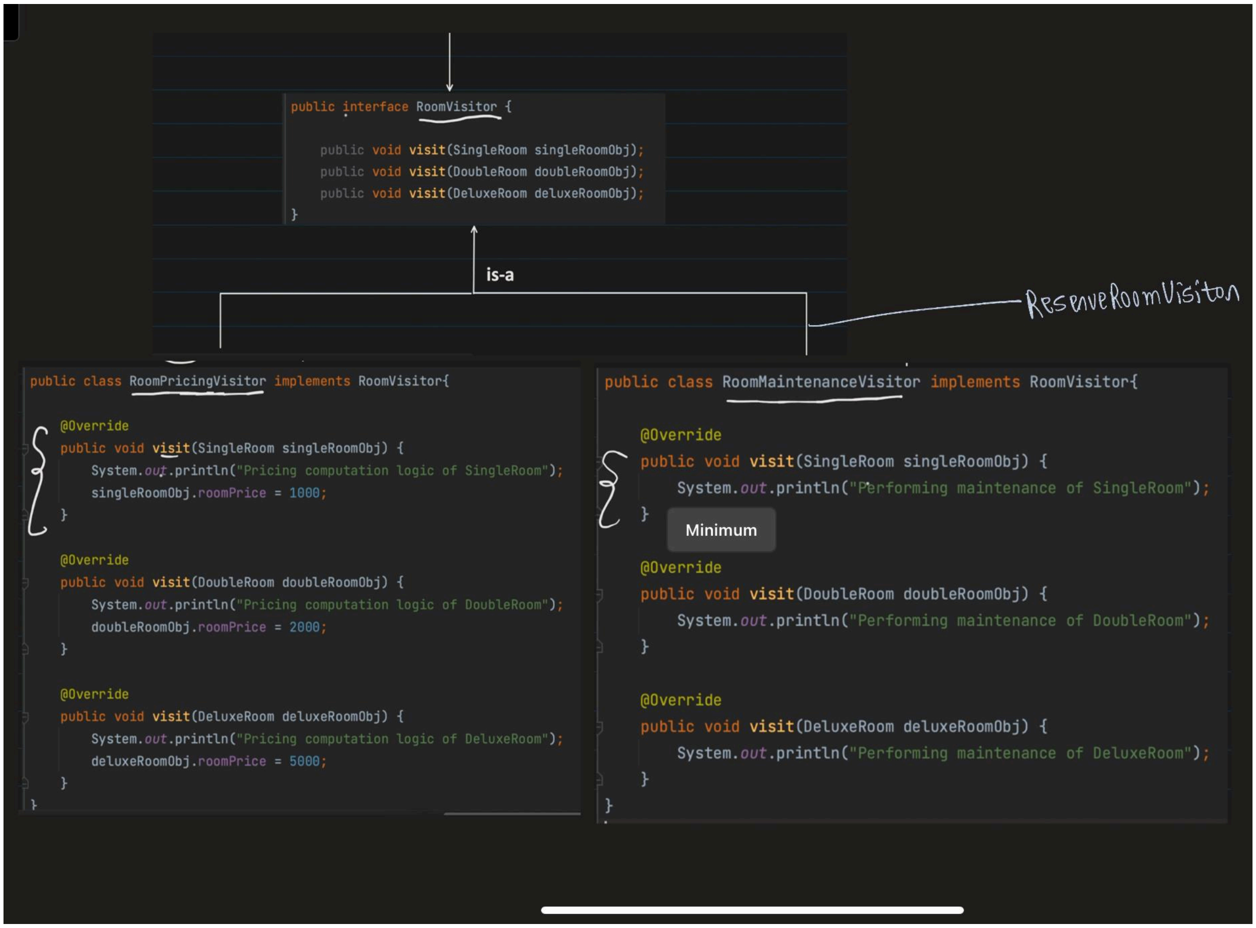
```java
public class DeluxeRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( deluxeRoomObj: this);
    }
}
```



```java
public interface RoomVisitor {

    public void visit(SingleRoom singleRoomObj);
    public void visit(DoubleRoom doubleRoomObj);
    public void visit(DeluxeRoom deluxeRoomObj);
}
```

```java
public class RoomPricingVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Pricing computation logic of SingleRoom");
        singleRoomObj.roomPrice = 1000;
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Pricing computation logic of DoubleRoom");
        doubleRoomObj.roomPrice = 2000;
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Pricing computation logic of DeluxeRoom");
        deluxeRoomObj.roomPrice = 5000;
    }
}
```

```java
public class RoomMaintenanceVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Performing maintenance of SingleRoom");
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Performing maintenance of DoubleRoom");
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Performing maintenance of DeluxeRoom");
    }
}
```



ReserveRoomVisitor

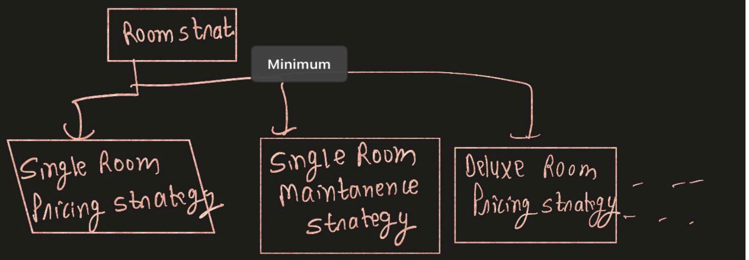
```java
public interface RoomVisitor {

    public void visit(SingleRoom singleRoomObj);
    public void visit(DoubleRoom doubleRoomObj);
    public void visit(DeluxeRoom deluxeRoomObj);
}
```

```java
public class RoomPricingVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Pricing computation logic of SingleRoom");
        singleRoomObj.roomPrice = 1000;
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Pricing computation logic of DoubleRoom");
        doubleRoomObj.roomPrice = 2000;
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Pricing computation logic of DeluxeRoom");
        deluxeRoomObj.roomPrice = 5000;
    }
}
```

```java
public class RoomMaintenanceVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Performing maintenance of SingleRoom");
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Performing maintenance of DoubleRoom");
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Performing maintenance of DeluxeRoom");
    }
}
```

* Double dispatch [illegible]
the method, which has to
be invoked, it's depend on
2 objects. 1 is Caller
& other is argument

* here with Strategy DP,


Room strat
Minimum
Single Room Pricing strategy
Single Room maintanence strategy
Deluxe Room Pricing strategy
- - -
- - -

(Wrong, these are operations strategy DP is mostly for seperating out algorithm, not operation, algo is independent)

(In Visitor, We are talking about operations)