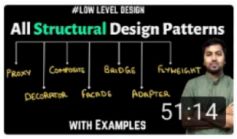
38 All behavioral Design Pattern

27. All Creational Design Patterns
Chapters: 00:00 - Introduction 00:50
Pattern 09:05 - Singleton Design Pat



32. All Structural Design Patterns
➡️ Notes: Shared in the Member Co
are Member of this channel, then pl

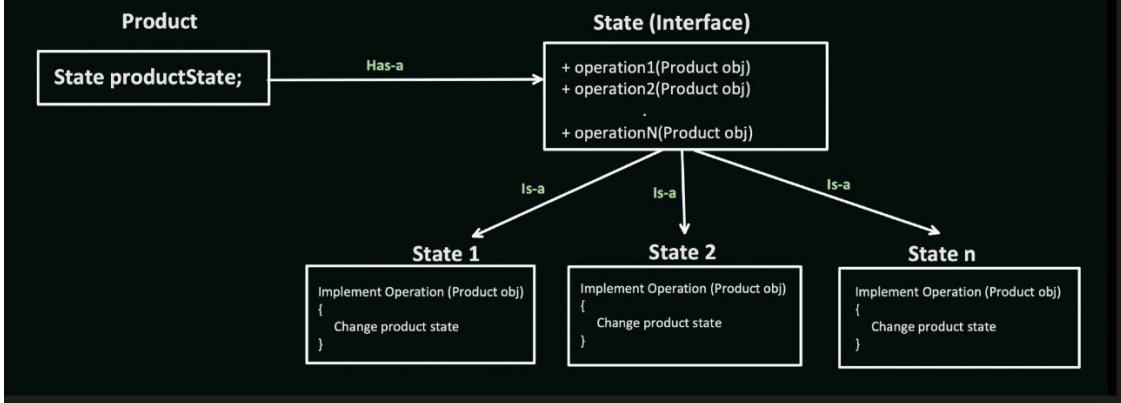
Behavioral Design Patterns:

Guides how different objects communicate with each other effectively and Distribute tasks efficiently, making software system flexible and easy to maintain.

1. State Pattern
2. Observer Pattern
3. Strategy Pattern
4. Chain of Responsibility Pattern
5. Template Pattern
6. Interpreter Pattern
7. Command Pattern
8. Iterator Pattern
9. Visitor Pattern
10. Mediator Pattern
11. Memento Pattern

Minimum

1. **State Pattern:** allows an object to alter its behaviour when its internal state changes.



Minimum

```java
public class VendingMachine {

    VendingState machineState;

    public VendingState getMachineState() {
        return machineState;
    }

    public void setMachineState(VendingState machineState) {
        this.machineState = machineState;
    }
}
```

Has-a

```java
public interface VendingState {

    void insertCoin(VendingMachine product);

    void dispenseItem(VendingMachine product);
}
```

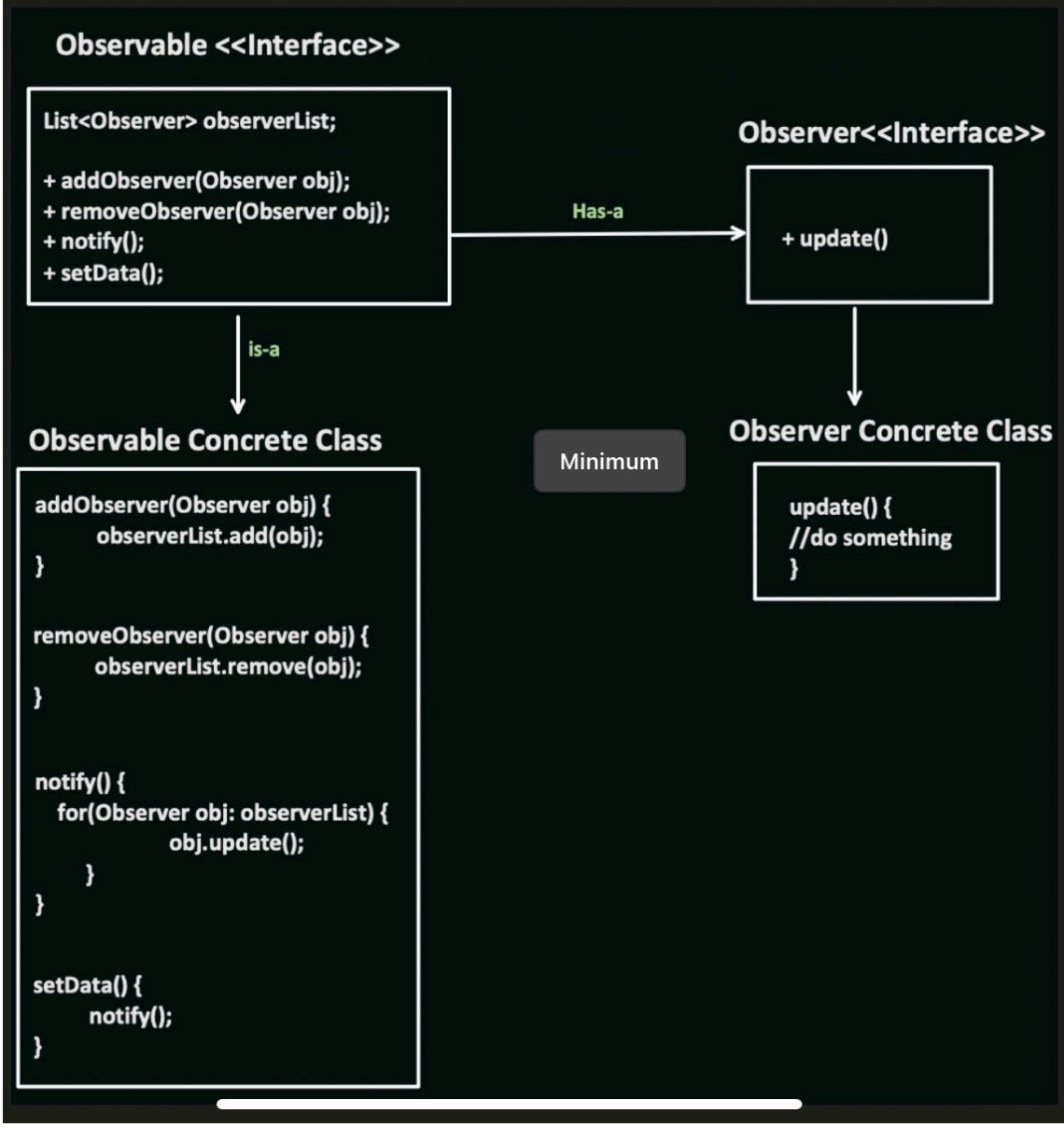
is-a

```java
public class IdleState implements VendingState{

    @Override
    public void insertCoin(VendingMachine product) {
        //insert coin logic
        System.out.println("Coin Inserted");
        product.setMachineState(new WorkingState());
    }

    @Override
    public void dispenseItem(VendingMachine product) {
        //not doing anything here
    }
}
```

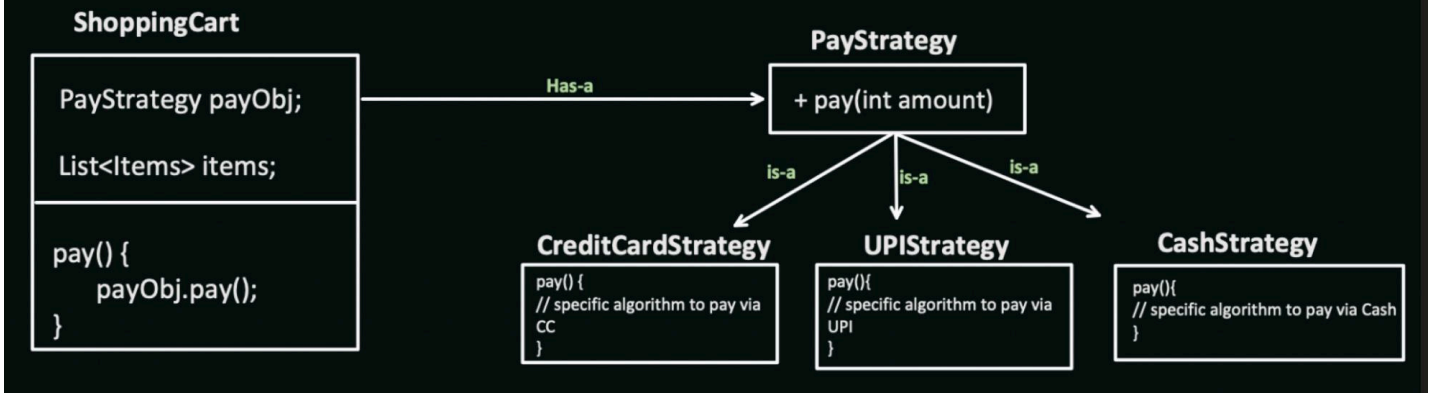
is-a

```java
public class WorkingState implements VendingState{

    @Override
    public void insertCoin(VendingMachine product) {
        //not doing anything here
    }

    @Override
    public void dispenseItem(VendingMachine product) {
        System.out.println("Product dispensed");
        //set any other state if applicable
    }
}
```

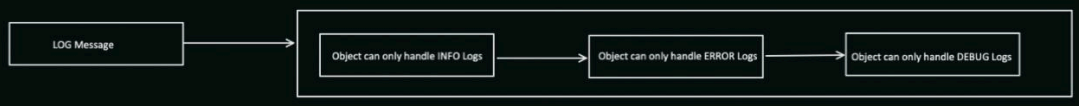
2. **Observer Pattern:** in this an object (Observable) maintains a list of its dependents (observers) and notifies them of any changes in its state.



3. **Strategy Pattern:** helps to define multiple algorithm for the task and we can select any algorithm depending on the situation.



4. **Chain of Responsibility Pattern:** allows multiple objects to handle a request without the sender needing to know which object will ultimately process it.



Client

```
LogProcessor obj = new InfoLogProcessor (new DebugLogProcessor(null)));

obj.log(some error message);
```

Minimum

LogProcessor

```
LogProcessor nextLogProcessor;

LogProcessor  (LogProcessor logProcessor) {
      nextLogProcessor = logProcessor;
}

void log(String message) {
      if(nextLogProcessor != null)
            nextLogProcessor.log(message);
}
```

Has-a

is-a

is-a

InfoLogProcessor

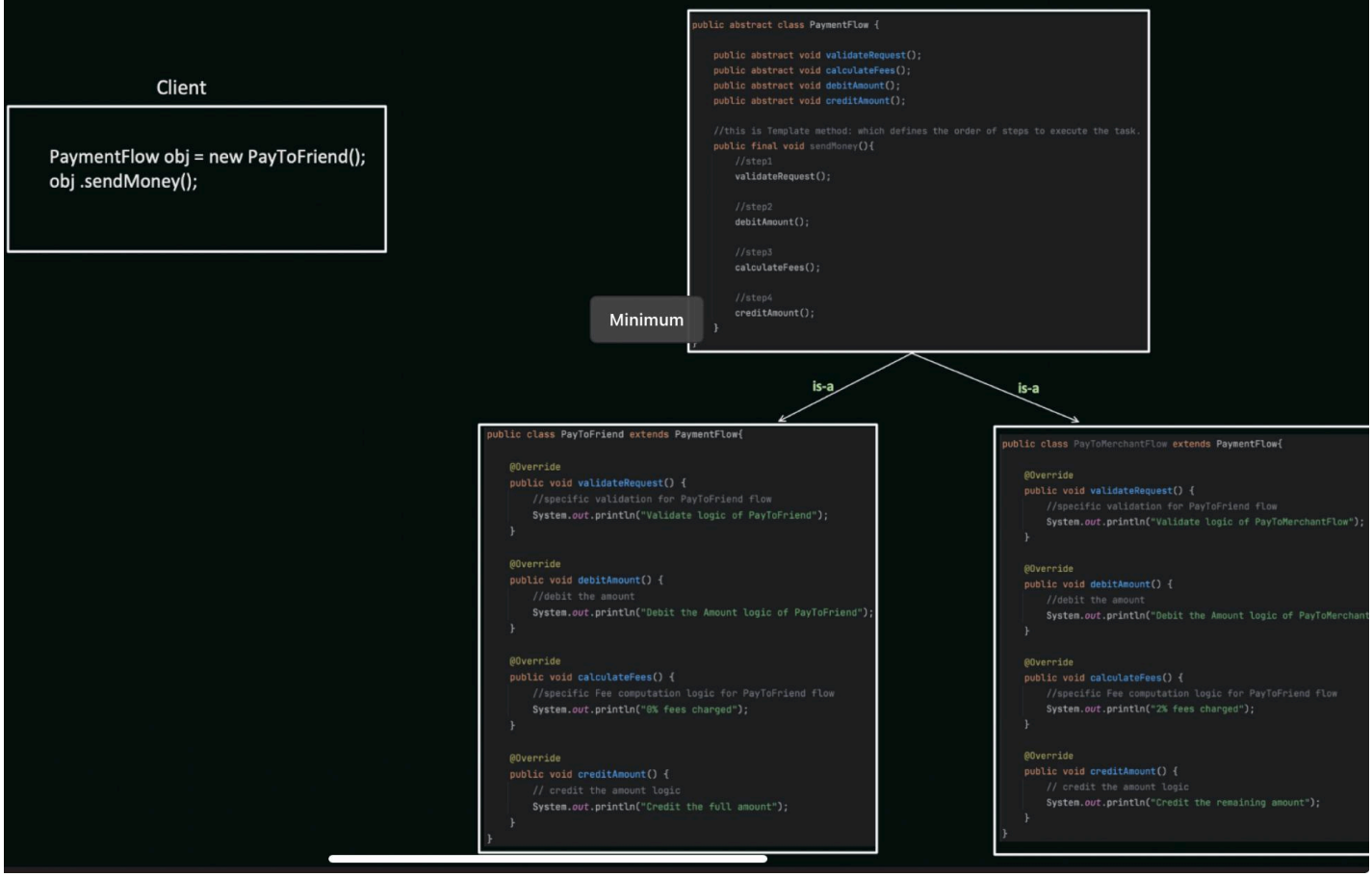
```
InfoLogProcessor (LogProcessor nextLogProcessor) {
      super(nextLogProcessor);
}

  void log(String message) {

      If(can i handle it) {
          // process this message
      }
      else {
          super.log(message);
      }
  }
```

→ Parent

DebugLogProcessor

```
DebugLogProcessor (LogProcessor nextLogProcessor) {
      super(nextLogProcessor);
}

  void log(String message) {

      If(can i handle it) {
          // process this message
      }
      else {
          super.log(message);
      }
  }
```

5. **Template Method Pattern:** when you want all classes to follow specific steps to process the tasks but provide flexibility that each class can have their own logic in that specific step.

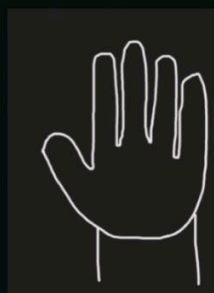



```java
public abstract class PaymentFlow {

    public abstract void validateRequest();
    public abstract void calculateFees();
    public abstract void debitAmount();
    public abstract void creditAmount();

    //this is Template method: which defines the order of steps to execute the task.
    public final void sendMoney(){
        //step1
        validateRequest();

        //step2
        debitAmount();

        //step3
        calculateFees();

        //step4
        creditAmount();
    }
}
```

```java
public class PayToFriend extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToFriend");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToFriend");
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("0% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the full amount");
    }
}
```

```java
public class PayToMerchantFlow extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToMerchantFlow");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToMerchant
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("2% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the remaining amount");
    }
}
```

6. Interpreter Pattern: Defines a grammar for interpreting and evaluating an expression.

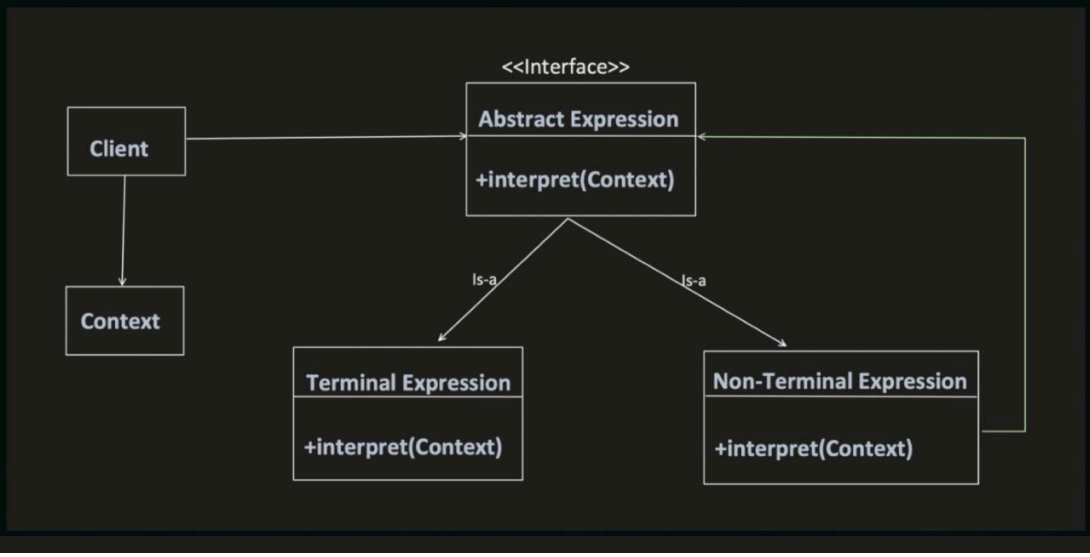


Some can interpret this:

- STOP
- Hi
- Number 5

Etc..

based on CONTEXT interpret happens.




```java
public class Client {

    public static void main(String args[]) {

        //initialize the context
        Context context = new Context();
        context.put( strVariable: "x",  intValue: 2);
        context.put( strVariable: "b",  intValue: 4);

        //exp
        AbstractExpression expression1 = new MultiplyNonTerminalExpression(
                new NumberTerminalExpression( stringVal: "a"), new NumberTerminalExpression( stringVal: "b"));

        System.out.println(expression1.interpret(context));
    }
}
```

```java
public interface AbstractExpression {

    int interpret(Context context);
}
```

Minimum

```java
public class Context {

    Map<String, Integer> contextMap = new HashMap<>();

    public void put(String strVariable, int intValue) {
        contextMap.put(strVariable, intValue);
    }

    public int get(String strVariable) {
        return contextMap.get(strVariable);
    }
}
```

```java
public class NumberTerminalExpression implements AbstractExpression{

    String stringValue;

    NumberTerminalExpression(String stringVal) {
        this.stringValue = stringVal;
    }

    @Override
    public int interpret(Context context) {
        return context.get(stringValue);
    }
}
```

```java
public class MultiplyNonTerminalExpression implements AbstractExpression{

    AbstractExpression leftExpression;
    AbstractExpression rightExpression;

    public MultiplyNonTerminalExpression(AbstractExpression leftExpression, AbstractExpression rightExpression) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }

    @Override
    public int interpret(Context context) {
        return leftExpression.interpret(context) * rightExpression.interpret(context);
    }
}
```

7. **Command Pattern:** Turns requests (commands) into objects, allowing you to either parametrized or queue them. This will help to decouple the request Sender and receiver.

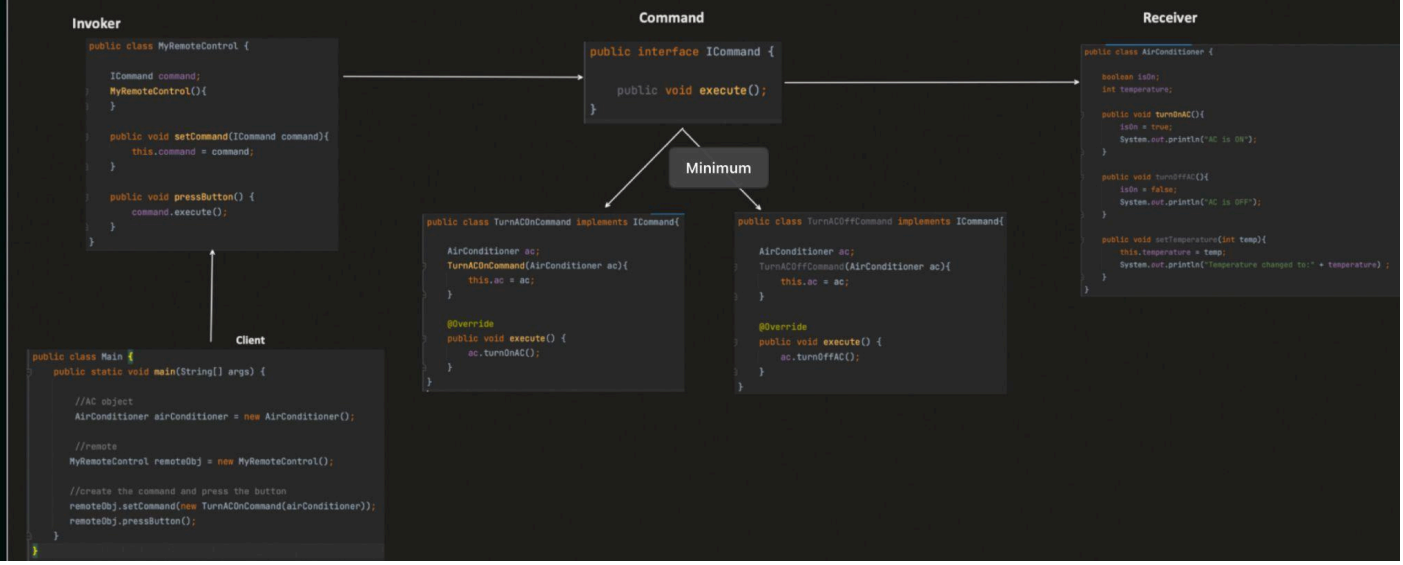
Problem with below code:

process of turning on AC is simple, but if there are more steps, client has to aware all of that, which is not good. So Sender and Receiver are not decoupled.

```java
public class Main {
    public static void main(String[] args) {

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();
    }
}
```

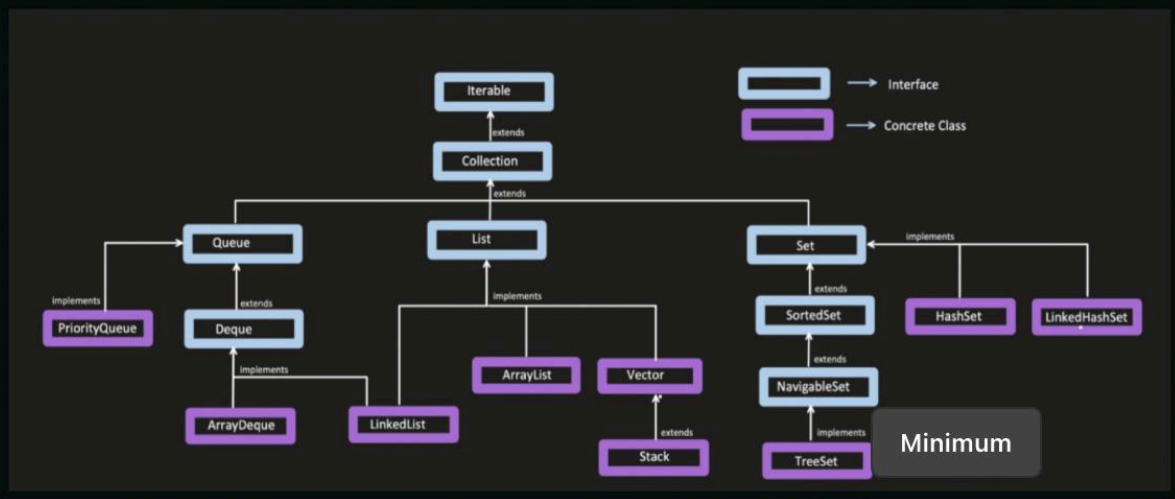
```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

Solution with Command Pattern:



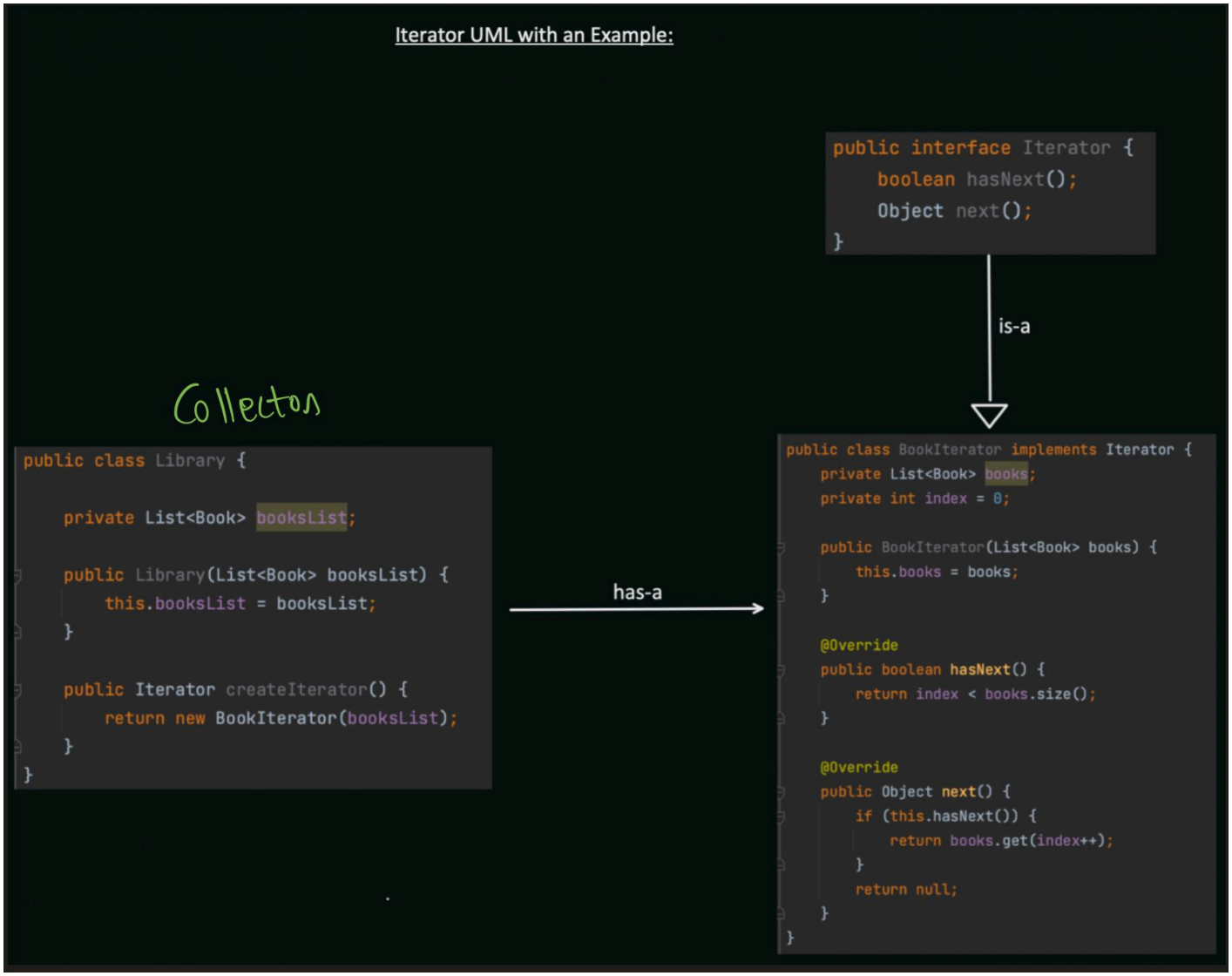
8. **Iterator Pattern:** that provides a way to access element of a Collection sequentially without exposing the underlying representation of the collection.

Understand the Need for an ITERATOR Pattern:




```java
public class LinkedHashSetExample {

    public static void main(String args[]){

        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

       Iterator<Integer> iterable =  intSet.iterator();
       while(iterable.hasNext()){
           int val = iterable.next();
           System.out.println(val);
       }
    }
}
```

this logic is same for every


Iterator UML with an Example:
public interface Iterator {
boolean hasNext();
Object next();
}
is-a
Collectors
public class Library {
private List<Book> booksList;
public Library(List<Book> booksList) {
this.booksList = booksList;
}
public Iterator createIterator() {
return new BookIterator(booksList);
}
}
has-a
public class BookIterator implements Iterator {
private List<Book> books;
private int index = 0;
public BookIterator(List<Book> books) {
this.books = books;
}
@Override
public boolean hasNext() {
return index < books.size();
}
@Override
public Object next() {
if (this.hasNext()) {
return books.get(index++);
}
return null;
}
}

```java
public class Client {

    public static void main(String[] args) {
        List<Book> booksList = Arrays.asList(
                new Book( price: 100,  bookName: "Science"),
                new Book( price: 200,  bookName: "Maths"),
                new Book( price: 300,  bookName: "GK"),
                new Book( price: 400,  bookName: "Drawing")
        );

        Library lib = new Library(booksList);
        Iterator iterator = lib.createIterator();

        while (iterator.hasNext()) {
            Book book = (Book) iterator.next();
            System.out.println(book.getBookName());
        }
    }
}
```

```java
public class Book {

    private int price;
    private String bookName;

    Book(int price, String bookName){
        this.price = price;
        this.bookName = bookName;
    }
    public int getPrice() {
        return price;
    }

    public String getBookName() {
        return bookName;
    }
}
```

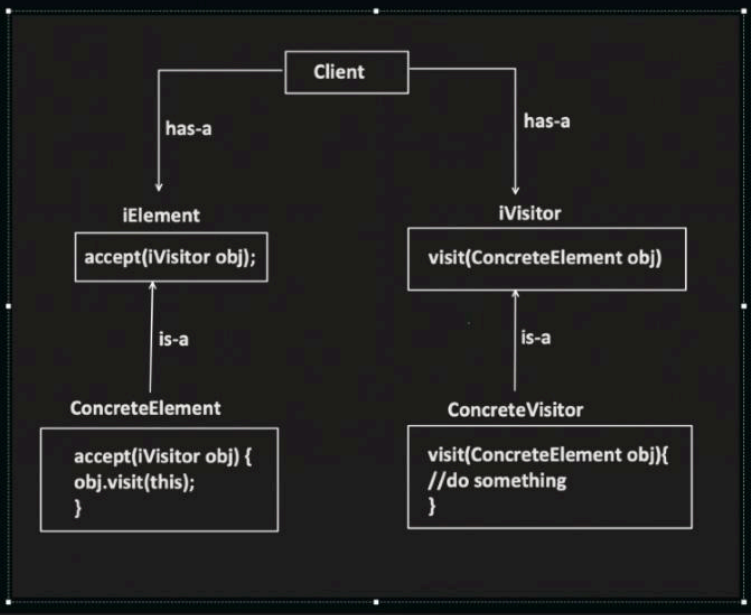
9. **Visitor Pattern:** Allows adding new operations to existing classes without modifying them and encourage OPEN/CLOSED principle.

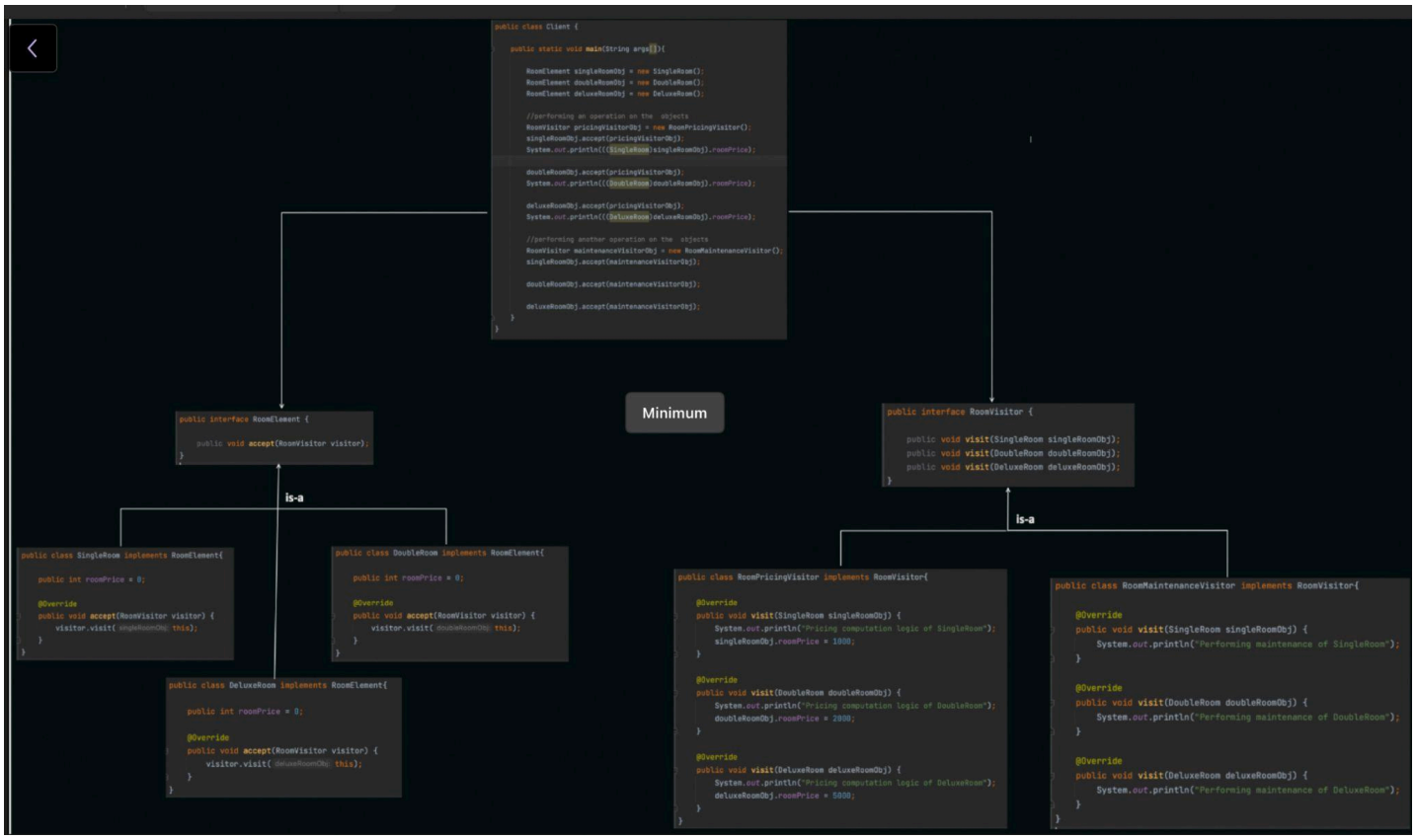
What's the problem with the below class?

```
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 operation                    the time
}
```

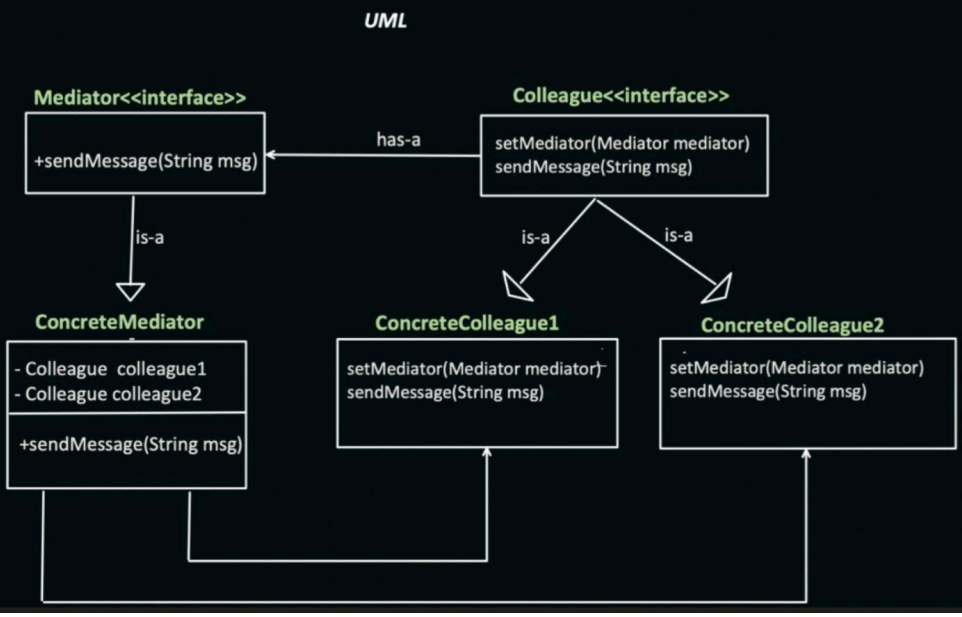
Minimum

UML of Visitor Pattern





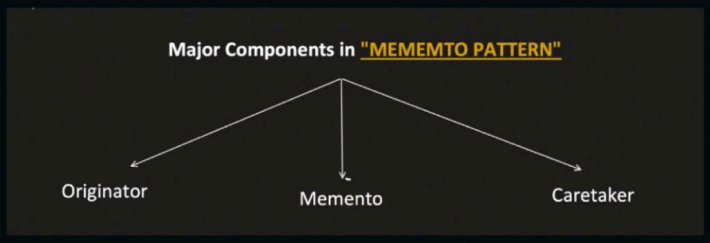
10. **Mediator Pattern:** It encourage loose coupling by keeping objects from referring to each other explicitly and allows them to ***communicate through a mediator*** object.




Lets use, Online Auction System Example to understand the UML
//this is Mediator Interface
public interface AuctionMediator {
void addBidder(Colleague bidder);
void placeBid(Colleague bidder, int bidAmount);
}
har - q
public interface Colleague {
void placeBid(int bidAmount);
void receiveBidNotification(int bidAmount);
String getName();
}
ui - q
Minimum
ui - q
/Mediator Concrete Class
ublic class Auction implements AuctionMediator {
List<Colleague> colleagues = new ArrayList<>();
@Override
public void addBidder(Colleague bidder) {
colleagues.add(bidder);
}
@Override
public void placeBid(Colleague bidder, int bidAmount) {
for(Colleague colleague : colleagues){
if(!colleague.getName().equals(bidder.getName())){
colleague.receiveBidNotification(bidAmount);
}
}
}
public class Bidder implements Colleague {
String name;
AuctionMediator auctionMediator;
Bidder(String name, AuctionMediator auctionMediator) {
this.name = name;
this.auctionMediator = auctionMediator;
auctionMediator.addBidder(this);
}
@Override
public void placeBid(int bidAmount) {
auctionMediator.placeBid(bidder: this, bidAmount);
}
@Override
public void receiveBidNotification(int bidAmount) {
System.out.println("Bidder: " + name + " got the notification that someone has put bid of : " + bidAmount);
}
@Override
public String getName(){
return name;
}
}

Memory

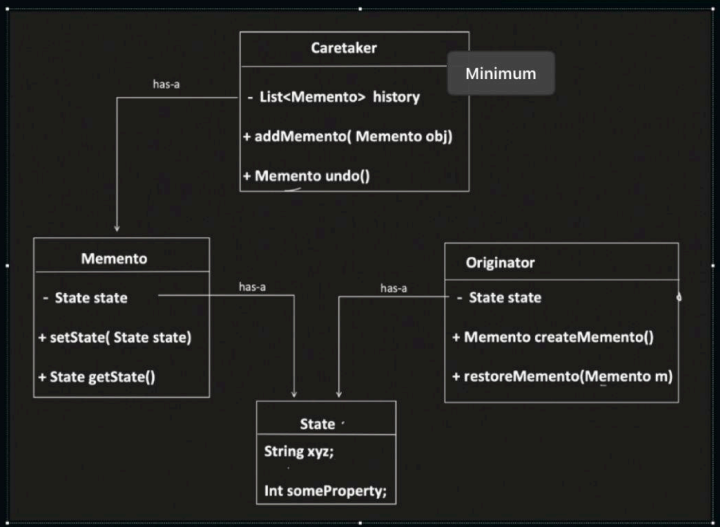
11. **Memento Pattern:** Provides an ability to revert an object to a previous state i.e. UNDO capability.
and it does not expose the object internal implementation.

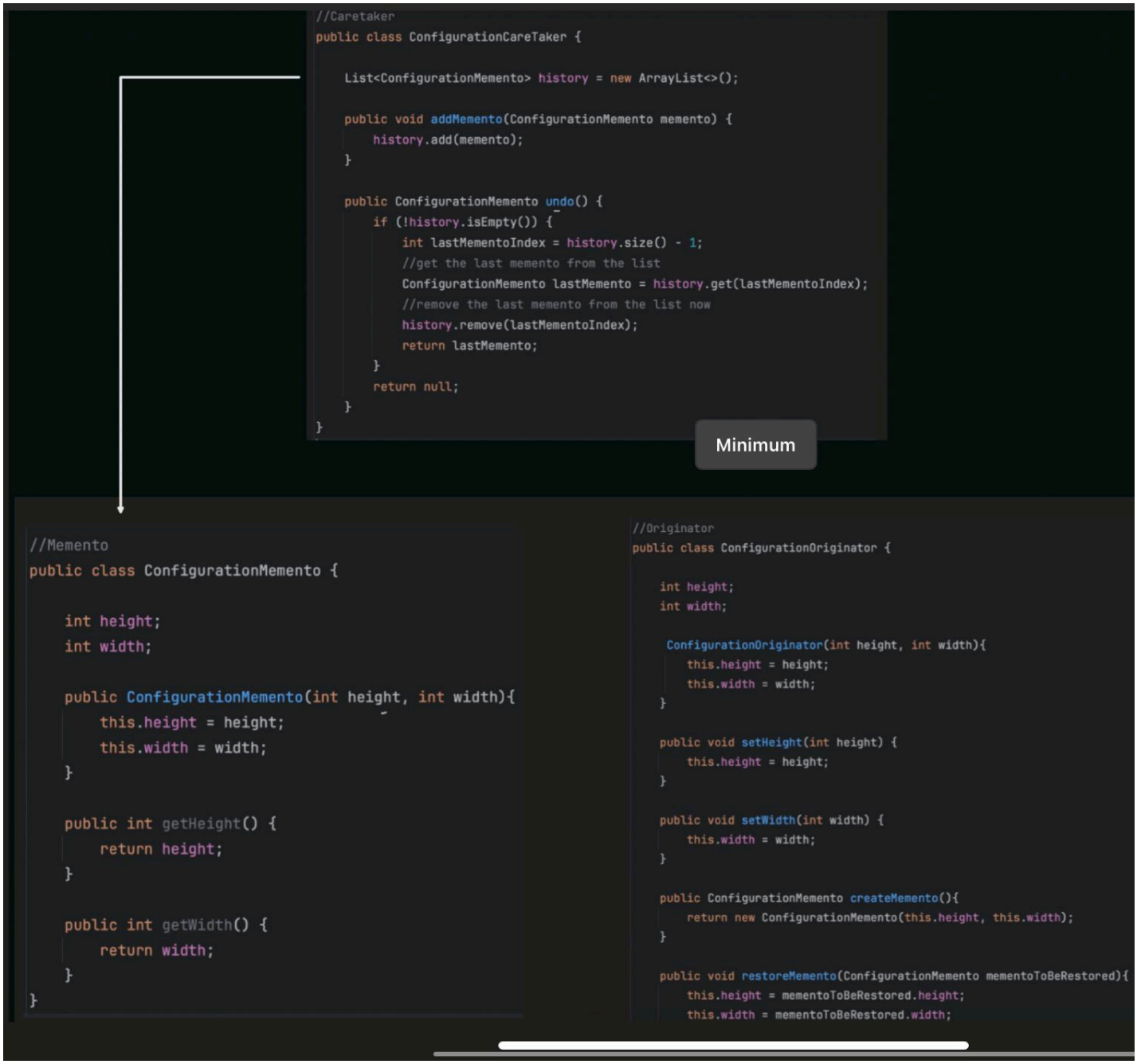


Originator:
- It represents the object, for which state need to be saved and restored.
- Expose Methods to Save and Restore its state using Memento object.

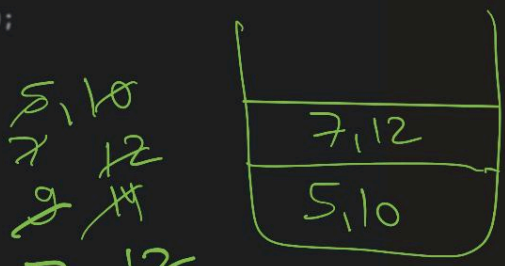
Memento:
- It represents an Object which holds the state of the Originator.

Caretaker:
- Manages the list of States (i.e. list of Memento)




//Caretaker
public class ConfigurationCareTaker {
List<ConfigurationMemento> history = new ArrayList<>();
public void addMemento(ConfigurationMemento memento) {
history.add(memento);
}
public ConfigurationMemento undo() {
if (!history.isEmpty()) {
int lastMementoIndex = history.size() - 1;
//get the last memento from the list
ConfigurationMemento lastMemento = history.get(lastMementoIndex);
//remove the last memento from the list now
history.remove(lastMementoIndex);
return lastMemento;
}
return null;
}
}
Minimum
//Memento
public class ConfigurationMemento {
int height;
int width;
public ConfigurationMemento(int height, int width){
this.height = height;
this.width = width;
}
public int getHeight() {
return height;
}
public int getWidth() {
return width;
}
}
//Originator
public class ConfigurationOriginator {
int height;
int width;
ConfigurationOriginator(int height, int width){
this.height = height;
this.width = width;
}
public void setHeight(int height) {
this.height = height;
}
public void setWidth(int width) {
this.width = width;
}
public ConfigurationMemento createMemento(){
return new ConfigurationMemento(this.height, this.width);
}
public void restoreMemento(ConfigurationMemento mementoToBeRestored){
this.height = mementoToBeRestored.height;
this.width = mementoToBeRestored.width;

```java
public class Client {

    public static void main(String args[]){

        ConfigurationCareTaker careTakerObject = new ConfigurationCareTaker();
        //initiate State of the originator
        ConfigurationOriginator originatorObject = new ConfigurationOriginator( height: 5,  width: 10);

        //save it
        ConfigurationMemento snapshot1 = originatorObject.createMemento();

        //add it to history
        careTakerObject.addMemento(snapshot1);

        //originator changing to new state
        originatorObject.setHeight(7);
        originatorObject.setWidth(12);

        //save it
        ConfigurationMemento snapshot2 = originatorObject.createMemento();

        //add it to history
        careTakerObject.addMemento(snapshot2);

        //originator changing to new state
        originatorObject.setHeight(9);
        originatorObject.setWidth(14);

        //UNDO
        ConfigurationMemento restoredStateMementoObj = careTakerObject.undo();
        originatorObject.restoreMemento(restoredStateMementoObj);

        System.out.println("height: " + originatorObject.height + " width: " + originatorObject.width );

    }
```



Minimum