`` Behavioral Design Pattern

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

Part - 1

1. Chain of Responsibility
2. Interpreter Pattern
3. Command Pattern
4. Iterator Pattern

Part- 2

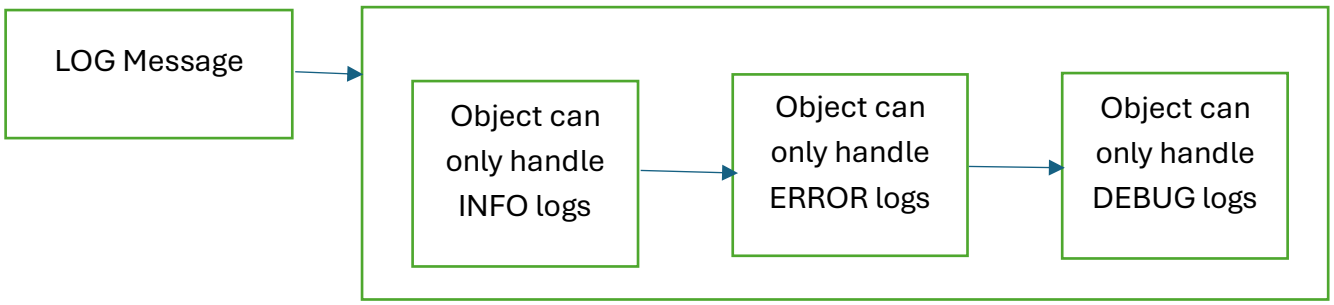
5. Mediator
6. Memento
7. Observer

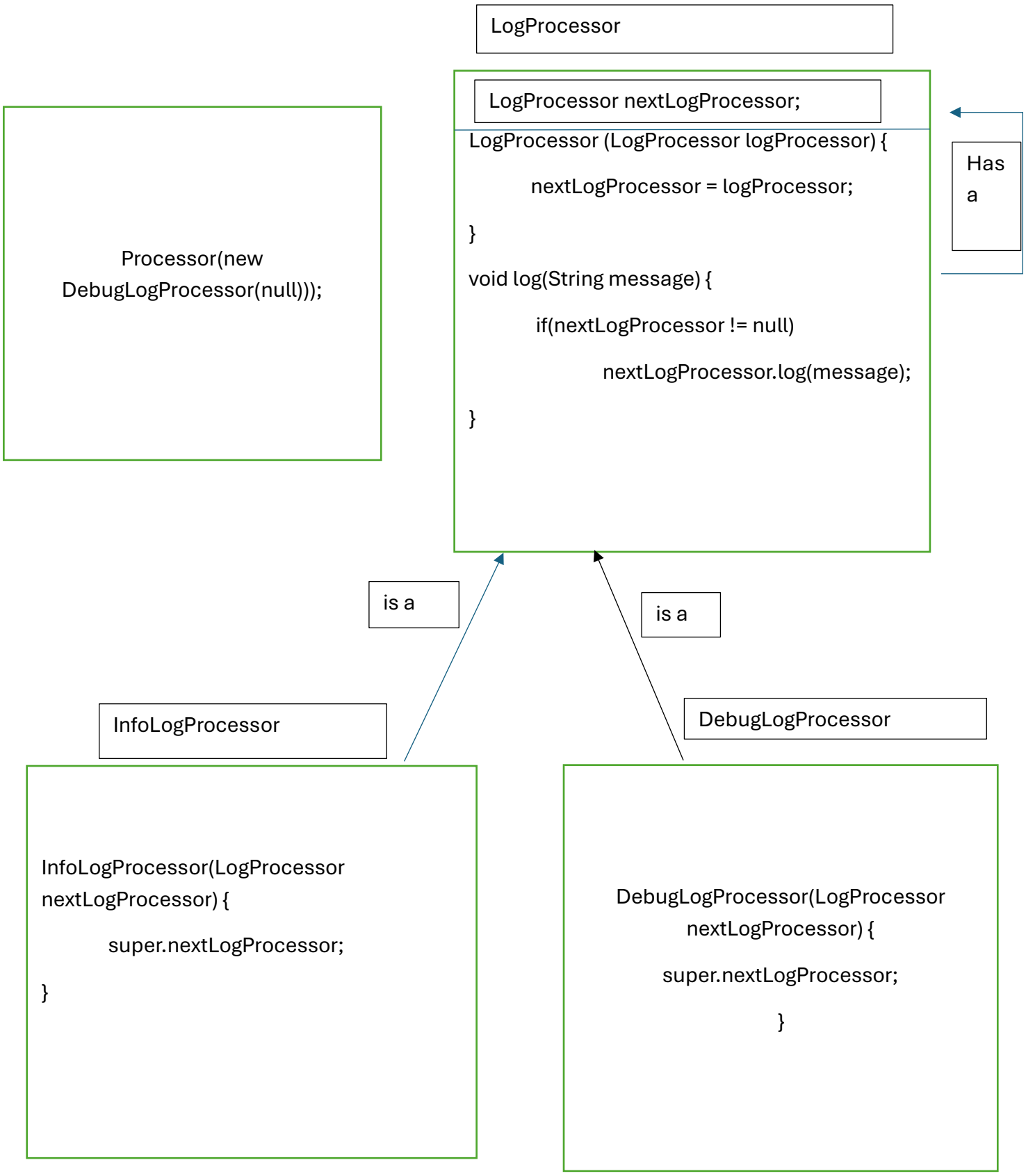
Part -3

8. State
9. Strategy
10. Template Method
11. Vistor

Chain of Responsibility Pattern:

Allows multiple objects to handle a request without the sender needing to know which object will ultimately process it .




LogProcessor
LogProcessor nextLogProcessor;
LogProcessor (LogProcessor logProcessor) {
nextLogProcessor = logProcessor;
}
void log(String message) {
if(nextLogProcessor != null)
nextLogProcessor.log(message);
}
Has a
Processor(new DebugLogProcessor(null)));
is a
is a
InfoLogProcessor
DebugLogProcessor
InfoLogProcessor(LogProcessor nextLogProcessor) {
super.nextLogProcessor;
}
DebugLogProcessor(LogProcessor nextLogProcessor) {
super.nextLogProcessor;
}

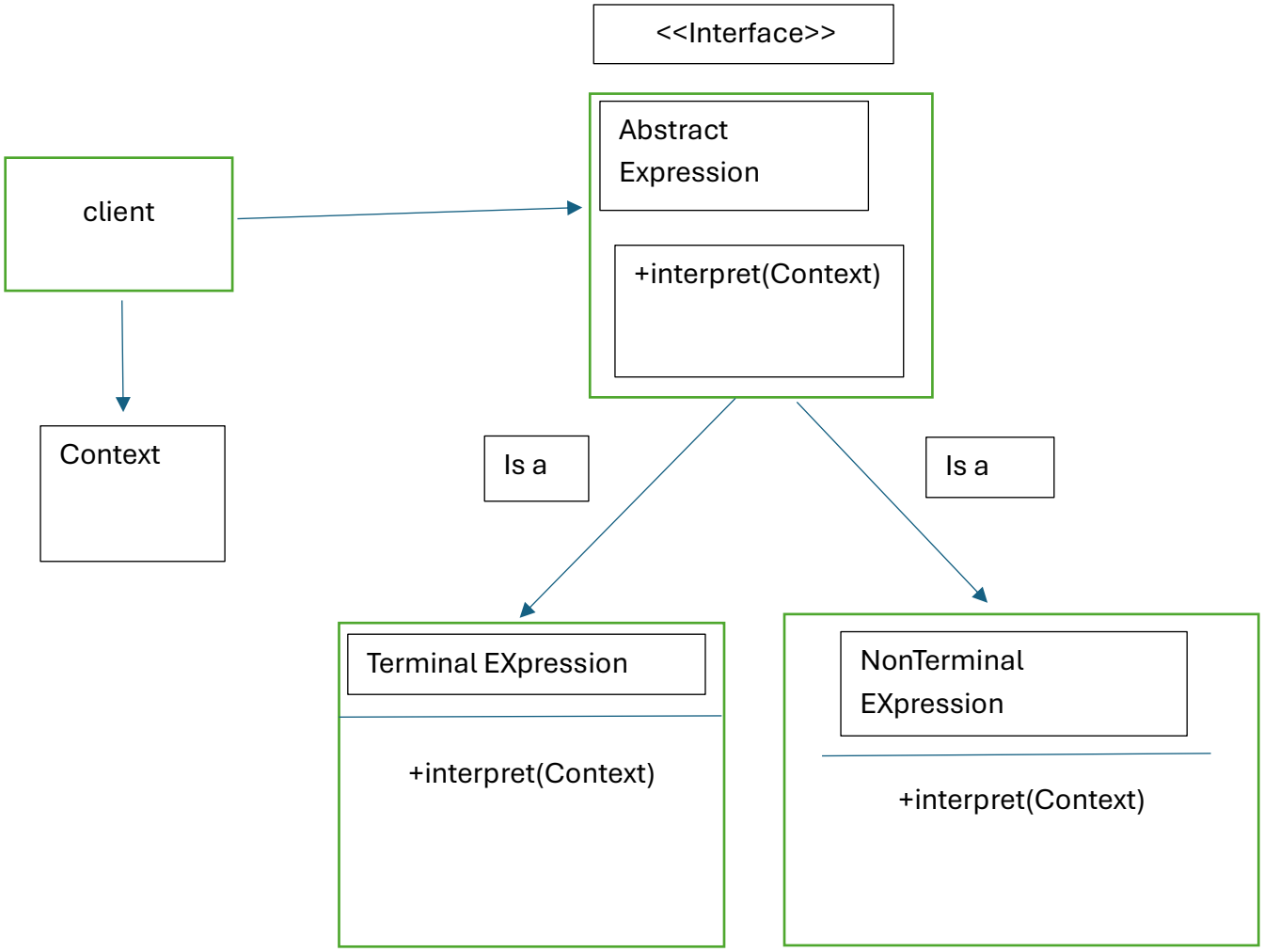
Interpreter Pattern:

Defines a grammar for interpreting and evaluating an expression

Some can interpret this :

- Stop
- Hi
- Number 5
- Esc...

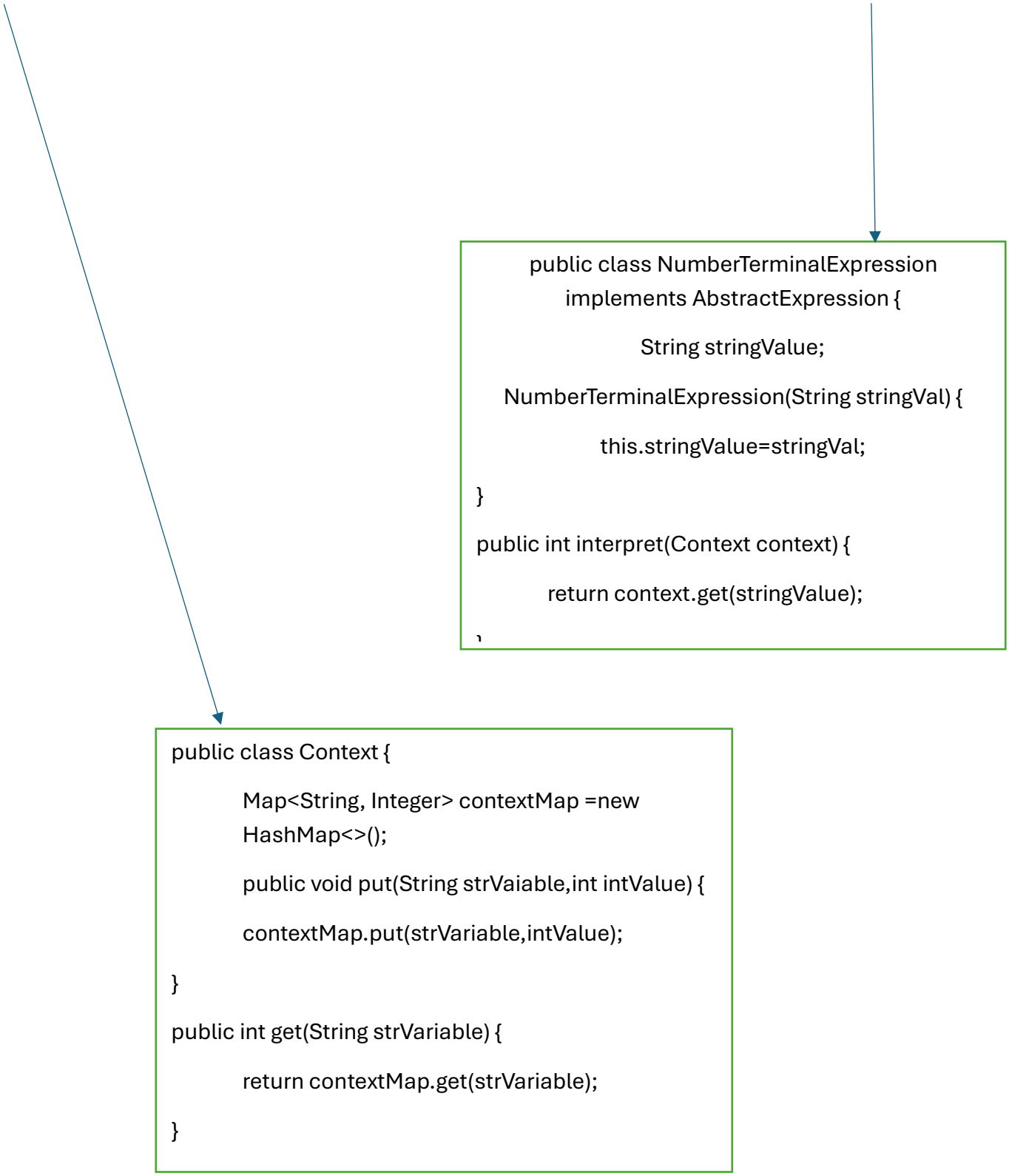
Based on CONTEXT interpret happens.



```
public class Client {

public static void main(String args[]) {

        //initialize the context

        Context context = new Context();

        context.put(strVariable “a”,intValue: 2);

        context.put(strVariable “b”,intValue :4);

//a*b

AbstractExpression expression1 = new
MultiplyNonTerminalExpression(new
NumberTerminalExpression(stringVal: “a”),new
NumberTerminalExpression(stringVal: “b”));

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

}
```

```
public interface
AbstractExpression {

int interpret(Context context);

 }
```

```
public class MultiplyNonTerminalExpression implements
AbstractExpression {

        AbstractExpression leftExpression;

        AbstractExpression rightExpression;

public MultiplyNonTerminalExpression(AbstractExpression
leftExpression, AbstractExpression rightExpression) {

        this. leftExpression = leftExpression;

        this. rightExpression= rightExpression;

}

public int interpret(Context context)  {

        return leftExpression.Interpret(context) *
rightExpression.interpret(context);

}
```


public class NumberTerminalExpression
implements AbstractExpression {
String stringValue;
NumberTerminalExpression(String stringVal) {
this.stringValue=stringVal;
}
public int interpret(Context context) {
return context.get(stringValue);
}
public class Context {
Map<String, Integer> contextMap =new
HashMap<>();
public void put(String strVaiable,int intValue) {
contextMap.put(strVariable,intValue);
}
public int get(String strVariable) {
return contextMap.get(strVariable);
}

(a*b) +(c*d)

```
//Abstract interface
public interface AbstractExpression {
int interpret(Context context);
 }

//multiply non terminal class
public class MultiplyNonTerminalExpression implements AbstractExpression {
        AbstractExpression leftExpression;
        AbstractExpression rightExpression;
public MultiplyNonTerminalExpression(AbstractExpression leftExpression,
AbstractExpression rightExpression) {
        this. leftExpression = leftExpression;
        this. rightExpression= rightExpression;
}
public int interpret(Context context)  {
        return leftExpression.Interpret(context) * rightExpression.interpret(context);
}
//sum non terminal class
public class SumNonTerminalExpression implements AbstractExpression {
        AbstractExpression leftExpression;
        AbstractExpression rightExpression;
public SumNonTerminalExpression(AbstractExpression leftExpression,
AbstractExpression rightExpression) {
        this. leftExpression = leftExpression;
        this. rightExpression= rightExpression;
}
```

```
public int interpret(Context context)  {
return leftExpression.Interpret(context) + rightExpression.interpret(context);
}
}
```

- **//Client code**

```
public class Client {
public static void main(String args[]) {
        //initialize the context
        Context context = new Context();
        context.put(strVariable “a”,intValue: 2);
        context.put(strVariable “b”,intValue :4);
        context.put(strVariable “c”,intValue: 8);
        context.put(strVariable “d”,intValue :10);
```

//(a*b) +(c*d)

```
AbstractExpression expressio2 = new Sum
NonTerminalExpression(new MultiplyNonTerminalExpression(new
NumberTerminalExpression(stringVal: “a”),new NumberTerminalExpression(stringVal:
“b”)), new NumberTerminalExpression(stringVal: “c”),new
NumberTerminalExpression(stringVal: “d”)));
System.out.println(expression2.interpret(context)); }
}
```

//context code

```
public class Context {
        Map<String, Integer> contextMap =new HashMap<>();
        public void put(String strVaiable,int intValue) {
        contextMap.put(strVariable,intValue);
}
```

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

We can optimize instead of writing multiplynonterminal,addnonterminal

```
public class BinaryNonTerminalExpression implements AbstractExpression {
        AbstractExpression leftExpression;
        AbstractExpression rightExpression;
        char operator;
public BinaryNonTerminalExpression (AbstractExpression leftExpression,
AbstractExpression rightExpression,char operator) {
        this. leftExpression = leftExpression;
        this. rightExpression= rightExpression;
        this.operator=operator;
}

public int interpret(Context context)  {
        switch(operator) {
                case '+' :
                return leftExpression. Interpret(context)  + rightExpression.
Interpret(context)  ;
                case '*' :
                return leftExpression. Interpret(context)  * rightExpression.
Interpret(context)  ;
                default:
                        return 0;
        }
}
```

//client code

```
public class Client {
public static void main(String args[]) {
        //initialize the context
        Context context = new Context();
        context.put(strVariable “a”,intValue: 2);
        context.put(strVariable “b”,intValue :4);
        context.put(strVariable “c”,intValue: 8);
        context.put(strVariable “d”,intValue :10);
```

//(a*b) +(c*d) ➔ ((a,b,*),(c,d,*),+)

```
AbstractExpression expressio2 = new Binary
NonTerminalExpression(new BinaryNonTerminalExpression(new
NumberTerminalExpression(stringVal: “a”),new NumberTerminalExpression(stringVal:
“b”),’*’), new BinaryNonTerminalExpression(new NumberTerminalExpression(stringVal:
“c”),new NumberTerminalExpression(stringVal: “d”),’*’),’+’);
System.out.println(expression2.interpret(context)); }
}
```

Command Design Pattern

Lets take the use-cse of remote control which can control various home appliances and with that lets understand this problem,then we go with design pattern.

```
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(“Temperatuere changes to:”+temperature);
          }
}
```

Client code

```
public class Main(){
        public static void main(String args[])
        {
            AirConditioner ac = new AirConditioner();
            ac.turnOnAC();
             ac.setTemperature(24);
             ac.turnOffAC();    }    }
```

Problem of implementation

They might have many sequences might be them to perform any action like turn on client has to have knowledge of sequence to call

Lack of Abstraction:

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

Undo/Redo Functionality:

What if I want to implement the undo/redo capability. how it will be handled.

Difficulty in code maintenance:

What if in future,we have to support more commands for more devices example Bulb,Ma

```
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("Temperatuere changes to:"+temperature);
          }
}

public class Bulb {

      boolean isOn;
       int temperature;
        public void  turnOnBulb() {
                isOn=true;
                System.out.println("Bulb is ON");
        }
        public void turnOffBulb() {
                isOn=false;
                System.out.println("Bulb is OFF");
        }
}
```

Client code

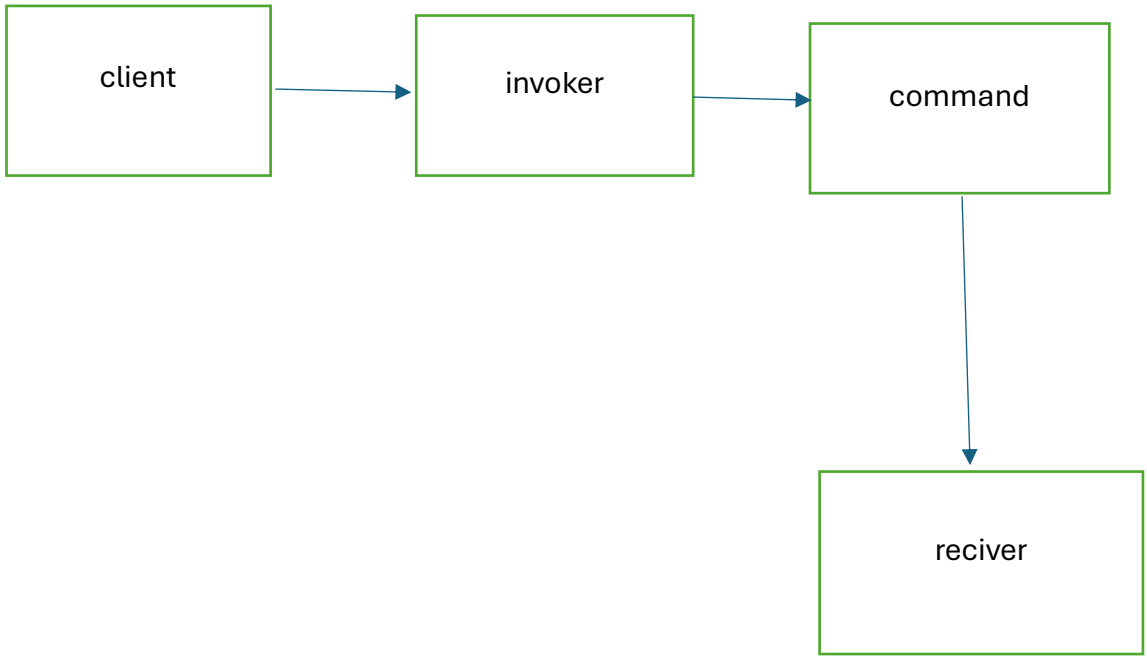
```
public class Main(){
            public static void main(String args[])
```

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

            Bulb bu = new Bulb();
             bu.turnOnBulb();
            bu.turnOffBulb();
            }
}
```

How Solved?

It divides logic into 3 parts

1. Receiver
2. Invoker and
3. Command



//Command

```
public interface ICommand
{
	public void execute();
}
 public class TurnACOnCommand implements ICommand {
	AirConditioner ac;
	TurnACOnCommand(AirConditioner ac) {
		this.ac=ac;
	public void execute() {
		ac.turnOnAC();
	}
	}
public class TurnACOffCommand implements ICommand {
	AirConditioner ac;
	TurnACOffCommand(AirConditioner ac) {
		this.ac=ac;
	}
	public void execute() {
		ac.turnOnAC();
	}
	}
```

//Invoker

```
	public class MyRemoteControl {
		ICommand command;
		MyRemoteControl() {
		}
```

```
        public void setCommand(ICommand command) {
            this.command=command;
        }
        public void pressButton() {
            command.execute();
    }
    }
```

//Client

```
public class Main {
    public static void main(String [] args) {
        AirConditioner airConditioner = new AirConditioner();
        MyRemoteControl remoteObj =new MyRemoteControl();
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();
    }
    }
```

//RECEVIER

```
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("Temperatuere changes to:"+temperature);
          }
}
```

//Receiver

```
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("Temperatuere changes to:"+temperature);
          }
}

public interface ICommand
{
          public void execute();
```

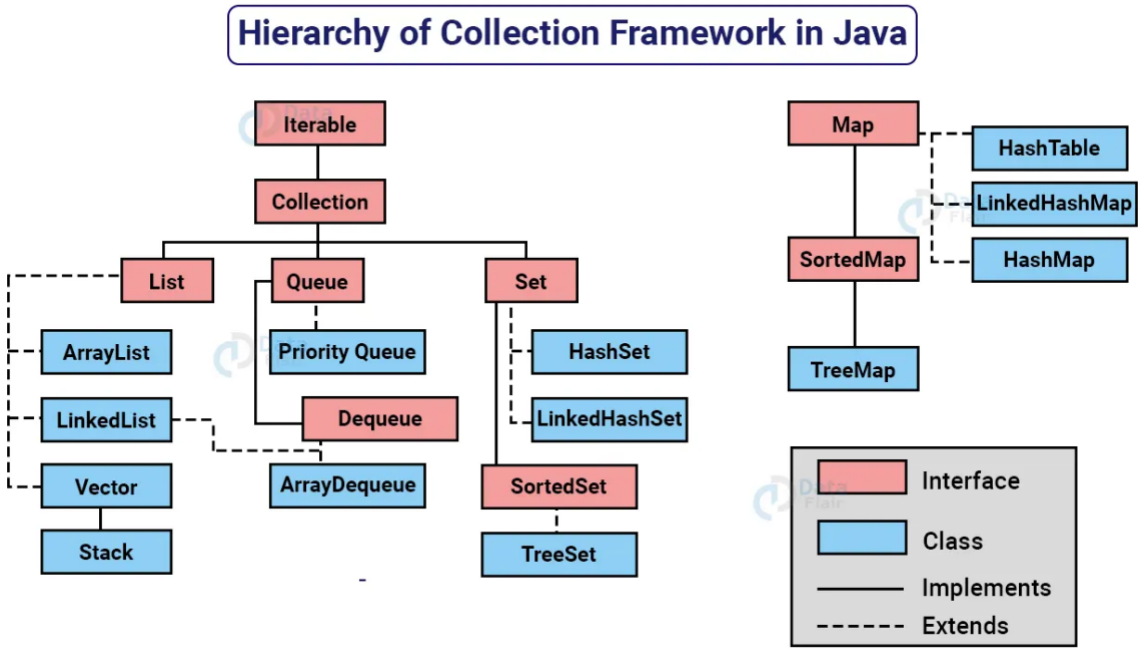
```
        public void undo();
}
 public class TurnACOnCommand implements ICommand {
        AirConditioner ac;
        TurnACOnCommand(AirConditioner ac) {
                this.ac=ac;
        public void execute() {
                ac.turnOnAC();
        }
        public void undo() {
                ac.turnOnAC();
        }
}
public class TurnACOffCommand implements ICommand {
        AirConditioner ac;
        TurnACOffCommand(AirConditioner ac) {
                this.ac=ac;
        }
        public void execute() {
                ac.turnOnAC();
        }
        public void undo() {
                ac.turnOffAC();
        }
        }
        //Invoker
        Import java.util.Stack;
        public class MyRemoteControl {
```

```
        Stack<ICommand> acCommandHistory = new Stack<>();
        ICommand command;
        MyRemoteControl() {
        }
        public void setCommand(ICommand command) {
            this.command=command;
        }
        public void pressButton() {
            command.execute();
    }
    public void undo() {
        if(!acCommandHistory.isEmpty() {
            ICommand lastCommand = acCommandHistory.pop();
            lastCommand.undo();
            }
        }
    }
```

//Client

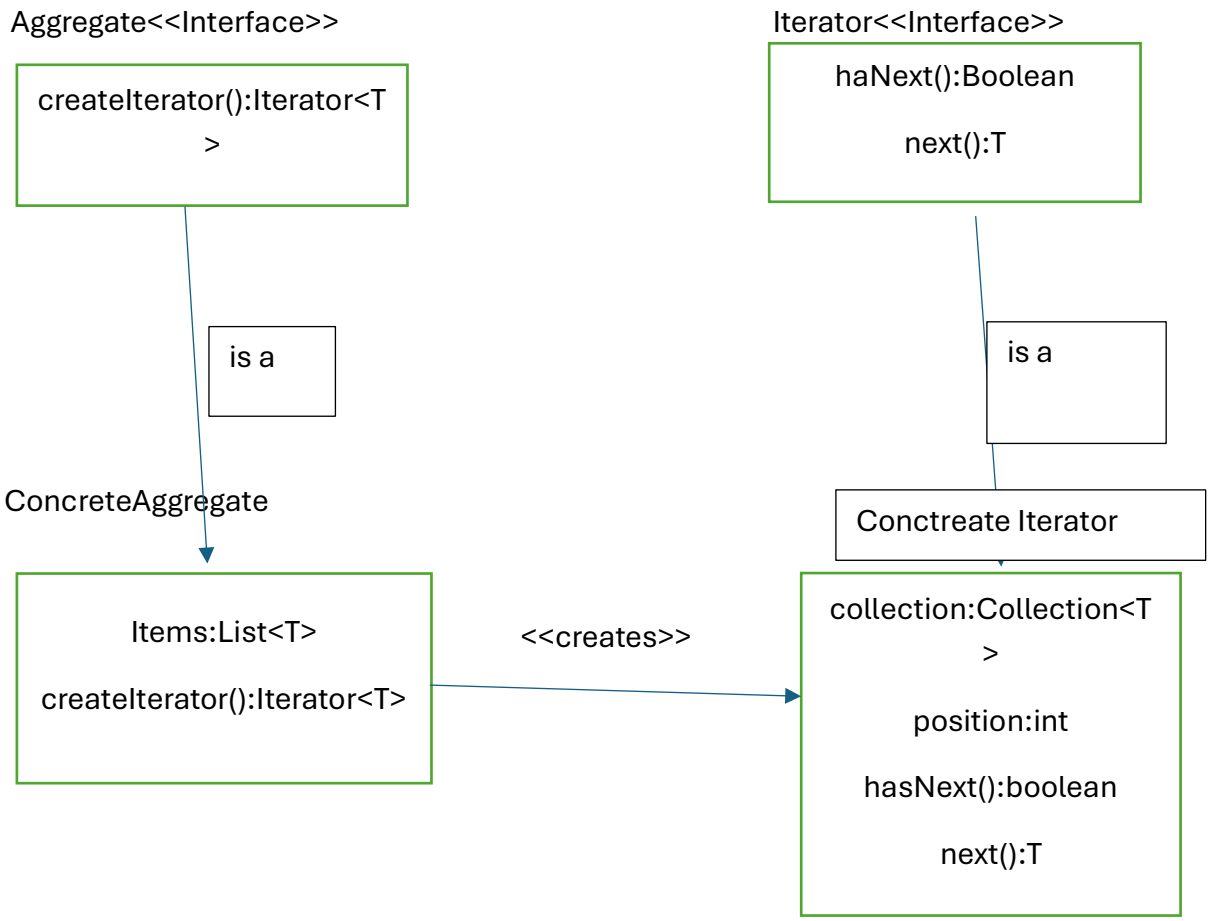
```
public class Main {
    public static void main(String [] args) {
        AirConditioner airConditioner = new AirConditioner();
        MyRemoteControl remoteObj =new MyRemoteControl();
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();
        remoteObj.undo();
    }
    }
```

Iterator Design pattern




```
public class LinkedHashSetExample {
        public static void main(String args[]) {
                Set<Integer> inset = new LinkedHashSet<>();
                intSet.add(2);
                intSet.add(77);
                intSet.add(82);
                intSet.add(63);
                intSet.add(5);
                Iterator<Integer> iterable = inSet.interator();
                while(iterable.hasNext() {
                        int val= iterable.next();
                        System.out.println(val);
                }
        }
}
```

So It's a Behavioral design pattern that provides a way to access elements of a collection sequentially without exposing the underlying representation of the collection.

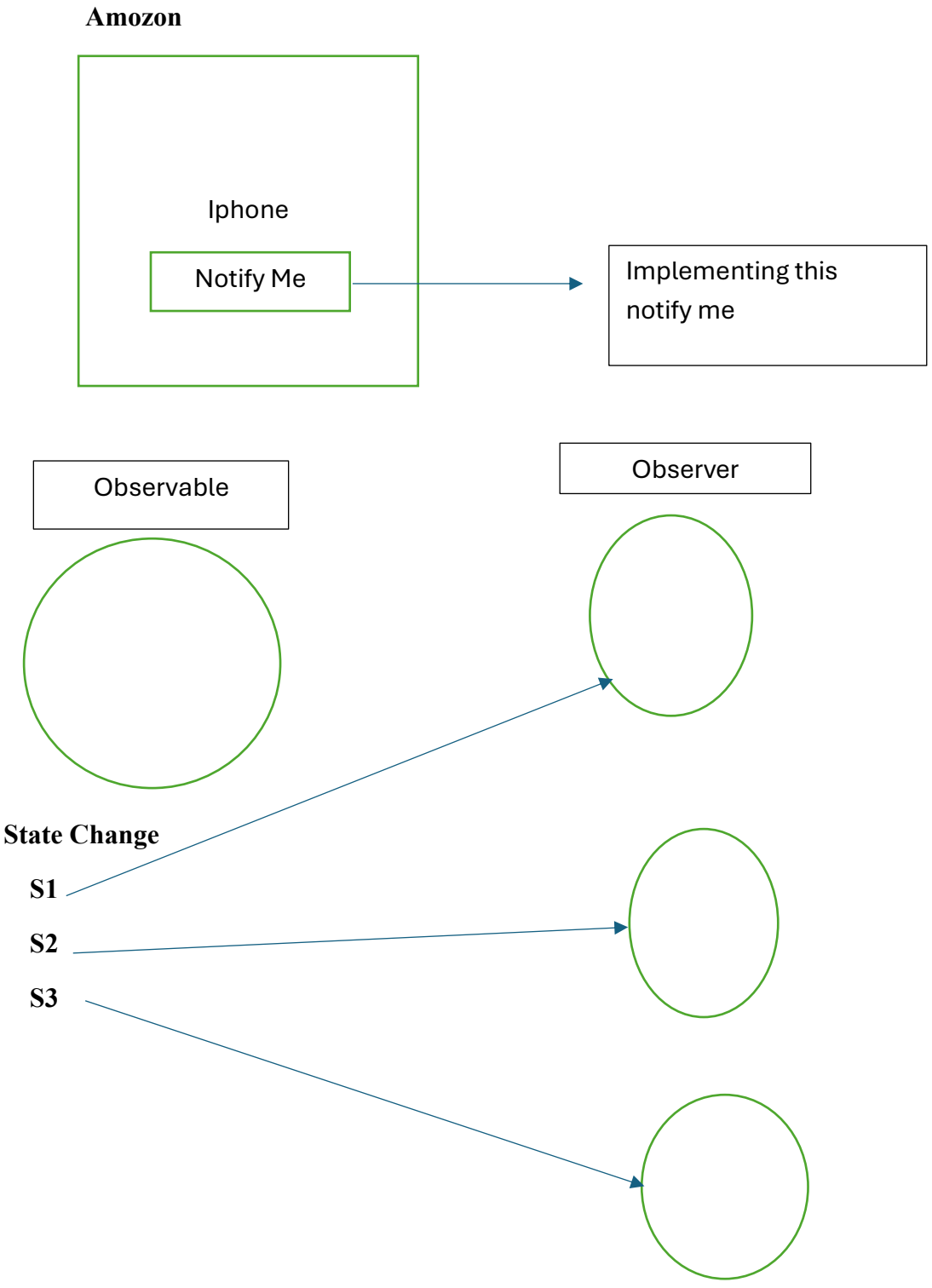



```
public interface Aggregate {
        Iterator createIterator();
}
public class Library implements Aggregate {
        private List<Book> bookList;
        public Library(List<Book> bookList) {
            this.booksList= bookList;
        }
        public Iterator createIterator() {
            return new BookIterator(booksList);
```

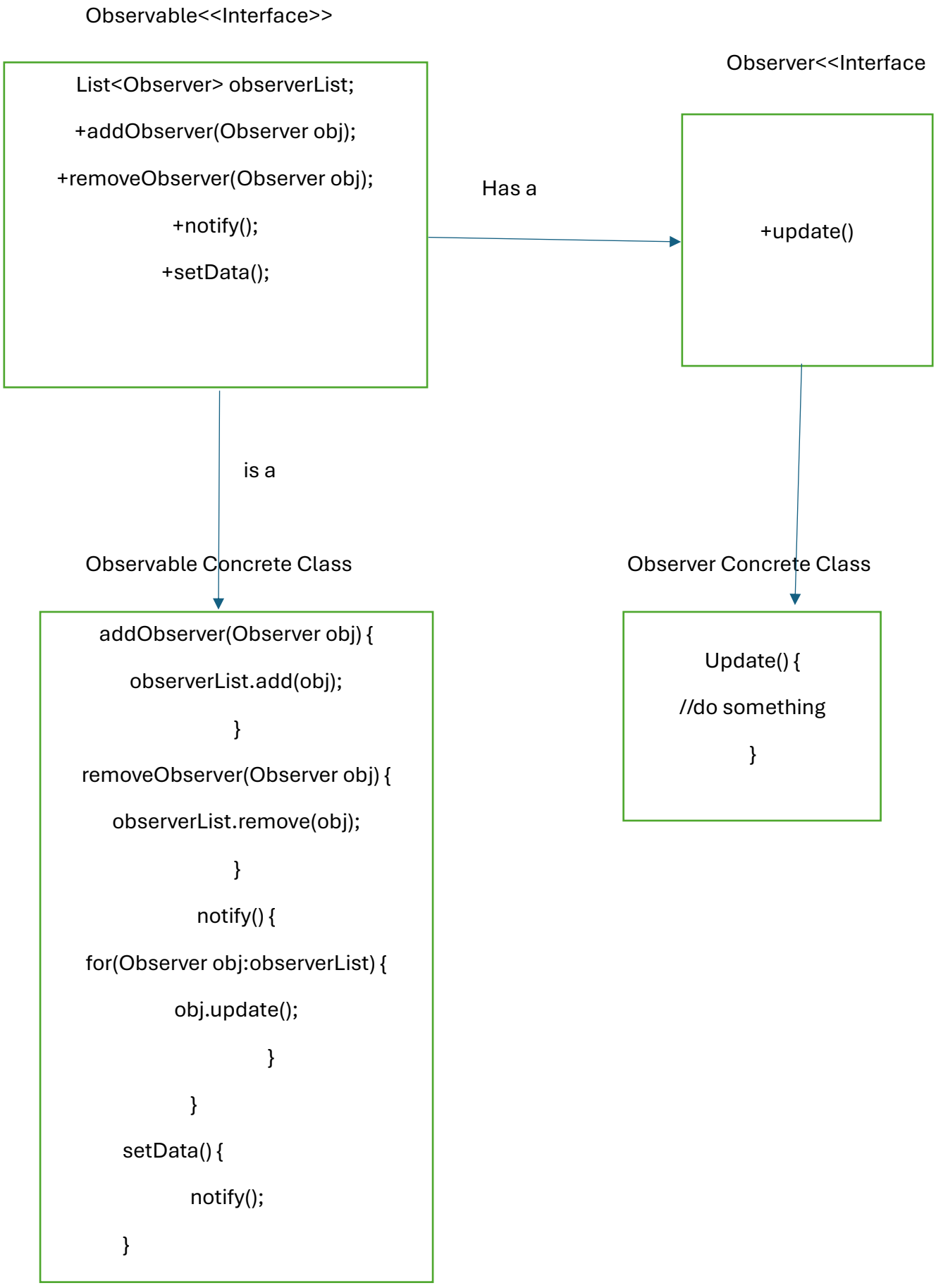
```
        }
}



public interface Iterator {
        boolean hasNext();
        Object next();
}
 public class BookIterator implements Iterator
{
        private List<Book> books;
        private int index=0;
        public BookIterator(List<Book> books) {
                this.books=books;
        }
        public boolean  hasnext() {
                return index < books.size();
        }
        public Object next() {
                if(this.hasNext()) {
                        return books.get(index++);
        }
        return null;
        }
}
public class Client {
        public static void main(String [] args) {
                List<Book> booksList = ArrayList(
```

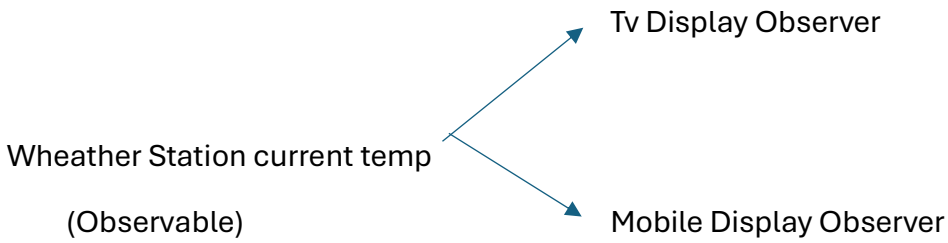
```
                    new Book(price:100,bookName:”Science”);
                    new Book(price:100,bookName:”Maths”);
                    new Book(price:100,bookName:”GK”);
                    new Book(price:100,bookName:”Drawing”);
        };
        Library lib= new library(booksList);
        Iterator iterator = lib.createIterator();
        while(iterator.hasNext())  {
            Book book = (Book) iterator.next();
            System.out.println(book.BookName());
        }
}
}
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;
}
}
```

Observer Design Pattern



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


Observable<<Interface>>
List<Observer> observerList;
+addObserver(Observer obj);
+removeObserver(Observer obj);
+notify();
+setData();
Has a
Observer<<Interface
+update()
is a
Observable Concrete Class
addObserver(Observer obj) {
observerList.add(obj);
}
removeObserver(Observer obj) {
observerList.remove(obj);
}
notify() {
for(Observer obj:observerList) {
obj.update();
}
}
setData() {
notify();
}
Observer Concrete Class
Update() {
//do something
}




```
WsObservable {
        add(DisplayObserver obj);
        remove(DisplayObserver obj);
        notify();
        setTemp();
}
WSObservableImplement {
        List<DisplayObserver> displayList;
        int temp;
        add()
        {

        }
        remove()
        {

        }
        notify() {
                for(DisplayObserver obj:displayList)
                {
                        Obj.update();
                }
```

```
        setData(int newTemp)
        {
            currentTemp=newTemp;
            notify();
        }
}

DisplayObserver() {
        update();
}
MobileDispalyObserver {

    WsObservable  obj;
    MobileDispalyObserver(
    WsObservable (O)
    {
        this.obj=O;
    }
    update()
    {
        Obj.getdata();
    }
}


    }
TvDispalyObserver {
}
```

Observable

```
package Observerpattern.Observable;
import Observerpattern.Observer.NotificationAlertObserver;
public interface StockObservable {
	public void add(NotificationAlertObserver observer);
	public void remove(NotificationAlertObserver observer);
	public void notifySubscribers();
	public void setStockCount(int newStockadded);
	public int getStockCount();
}
```

Concrete Observable

```
package Observerpattern.Observable;
import Observerpattern.Observer.NotificationAlertObserver;
import java.util.ArrayList;
import java.util.List;
public class IphoneObservableImpl implements StockObservable {
	public List<NotificationAlertObserver> observerList = new ArrayList();
	public int stockCount=0;
	public void add(NotificationAlertObserver observer) {
		observerList.add(observer);
	}
	public void remove( NotificationAlertObserver observer) {
		observerList.remove(observer);
	}
	public void notifySubscribers()
	{
		for(NotificationAlertObserver  observer : observerList) {
```

```
                observer.update();
            }
        }
        public void setStockCount(int newStockAdded) {
            if(stockCount==0) {
                notifySubscribers();
            }
            stockCount=stockCount  + newStockAdded;
            }
            public int getStockCount() {
                return stockCount;
            }
}
```

Observer

```
package ObserverPattern.Observer;
public interface NotificationAlertObserver {
            public void update();          }
```

Concrete Observer

```
package Observerpattern.Observer;
import Observerpattern.Observable.StockObservable;
public class EmailAlertObserverImpl implements NotificationAlertObserver {
        String emailId;
        StockObservable observable;
        public EmailAlertObserverImpl(String emailId,StockObservable observable)
        {
            this.observable= observable;
            this.emailId= emailId;
        }
```

```
        public void update() {
                sendMail(emailId,"Product is in stock hurry up!");
        }
        private void sendMail(String emailId,String msg)
        {
                System.out.println("email sent to :" + emailId);
        }
}
```

Client

```
Package ObserverPattern;
import Observerpattern.Observable.IphoneObservableImpl;
import Observerpattern.Observable.StockObservable;
import Observerpattern.Observable.EmailAlertObservableImpl;
import Observerpattern.Observable.MobileAlertObservableImpl
import Observerpattern.Observer.NotificationAlertObserver;
public class Store {
        public static void main(String args[])
        StockObservable iphoneStockObservable =new  IphoneObservableImpl();

        NotificationAlertObserver observer1= new
EmailAlertObservableImpl("xyz@gmail.com",iphoneStockObservable);
NotificationAlertObserver observer2= new
EmailAlertObservableImpl("xyz@gmail.com",iphoneStockObservable);
NotificationAlertObserver observer3= new
EmailAlertObservableImpl("xyz@gmail.com",iphoneStockObservable);
iphoneStockObservable.add(observer1);
iphoneStockObservable.add(observer2);
```

```
iphoneStockObservable.add(observer3);
iphoneStockObservable.setStockCount(10);
        }
}
```

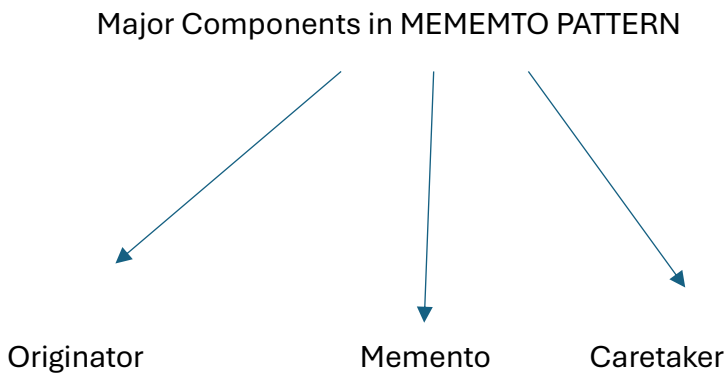
Memento Design Pattern or Snapshot design pattern

Why its required and When to use:

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

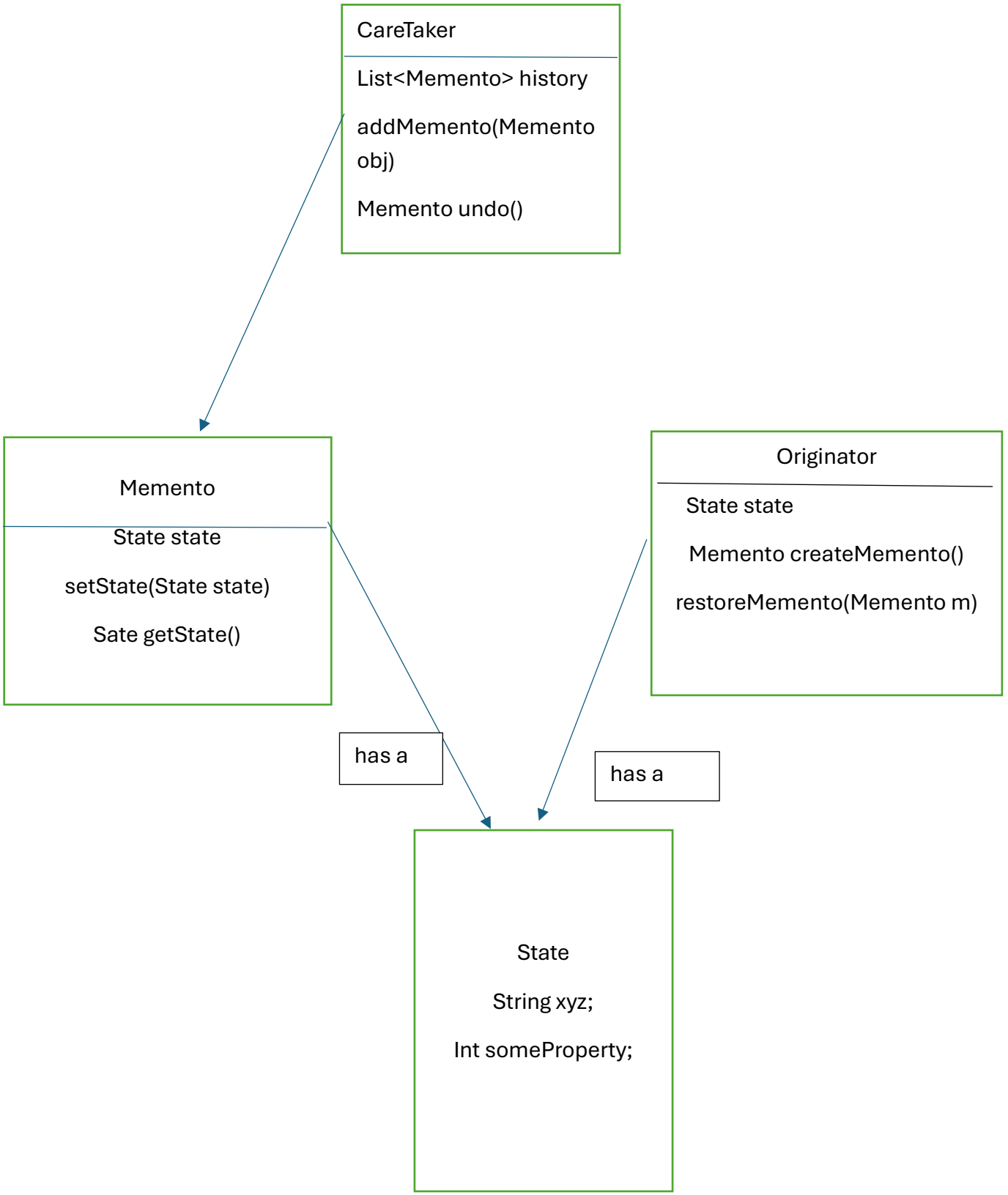
- It represents the object, for which state need to be saved and restored.
- Expose Methods to Save and Restore its state using Memento Object.

Memento :

- It represents an object which holds the state of the originator.

Caretaker:

- Manges the list of States (i.e list of Memento)


CareTaker
List<Memento> history
addMemento(Memento obj)
Memento undo()
Memento
State state
setState(State state)
Sate getState()
Originator
State state
Memento createMemento()
restoreMemento(Memento m)
has a
has a
State
String xyz;
Int someProperty;

```
//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(ConfiguartionMemento mementoToBeRestored) {
               this.height= mementoToBeRestored.height;
               this.width= mementoToBeRestored.width;
     }
}
//Memento
public class ConfigurationMemento  {
         int height;
          int width;
```

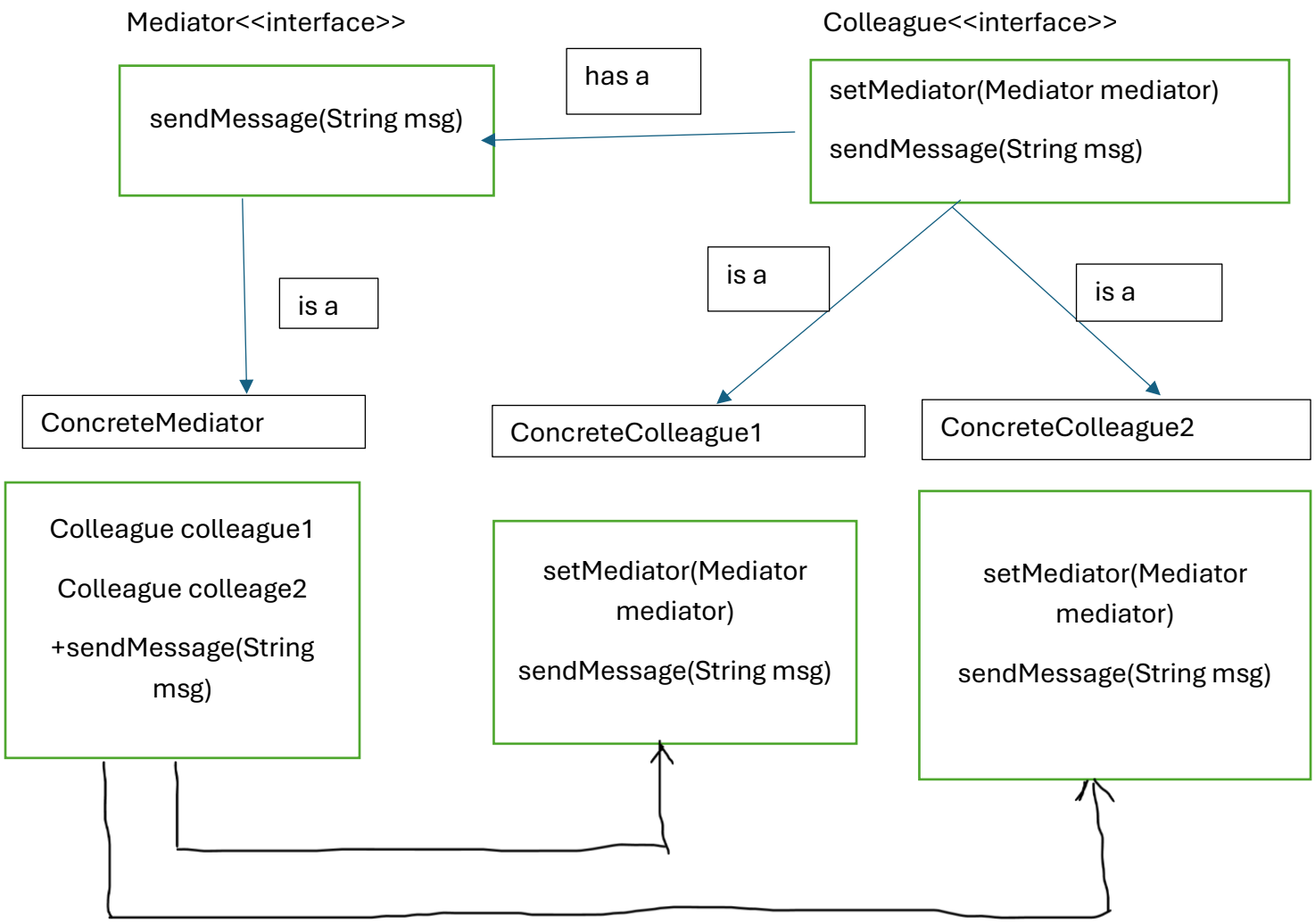
```
public ConfigurationMemento(int height,int width) {
        this.height= height;
        this.width= width;
}
public int getHeight() {
        return height;
}
public int getWidth() {
        return Width;
}
}
//CareTaker
public class ConfigurationCareTaker {
        List<ConfigurationMemento> history = new ArrayList<>();
        public void addMemento(ConfigurationMemento memento) {
                history.add(memento);
        }
       public ConfigurationMemento undo() {
          if(!history.isEmpty()) {
                  int lastMementotoIndex = 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;
}
```

```
//Client
public  class Client {
      public static void main(String args[]) {
           ConfigurationCareTaker     careTakerObject = new ConfigurationCareTaker();
            //initiate Sate of the originator
          ConfigurationMemento  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 restoreStateMementoObj = careTakerObject.undo();
        originatorObject.restoreMemento(restoredStateMementoObj);
      System.out.println("height:"  +  originatorObject.height  +  "width:"  +
originatorObject.width);
}
}
```

Mediator Design Pattern

The Mediator Pattern is a behavioral design pattern

It encourages loose coupling by keeping objects from referring to each other explicitly and allow them to communicate through a mediator object.



Lets see,Online Auction System Example to understand the UML

```
//Colleague Interface
public interface Colleague {
        void placeBid(int bidAmount);
```

```
        void receiveNotification(int bidAmount);
        String getName();
}
public class Bidder implements Colleague {
        String name;
        AuctionMediator auctionMediator;
        Bidder(String name,AuctionMediator auctionMediator) {
                this.name= name;
                this.auctionMediator = auctionMediator;
        }
        @override
        public void placeBid(int bidAmount) {
                auctionMediator.placeBit(bidder: this.bidAmount);
        }
        @override
        public void receiveBidNotification(int bidAmount) {
                System.out.println("Bidder: " + new + " get the notification that some one
has put bid of : " +bidAmount);
        }
        @override
        public String getName() {
                return name;
        }
}
//Main Class
public class Main {
        public static void main(String args[]) {
                Auctionmediator auctionMediatorobj = new Auction();
```

```
        Colleague bidder1 = new Bidder(name: ”A” ,auctionMediatorObj);
        Colleague bidder2 = new Bidder(name: ”B” ,auctionMediatorObj);
        bidder1.placeBid(bidAmount:2000);
        bidder2.placeBid(bidAmount:3000);
        bidder1.placeBid(bidAmount:3001);
        }
}
```

//Mediator Interface

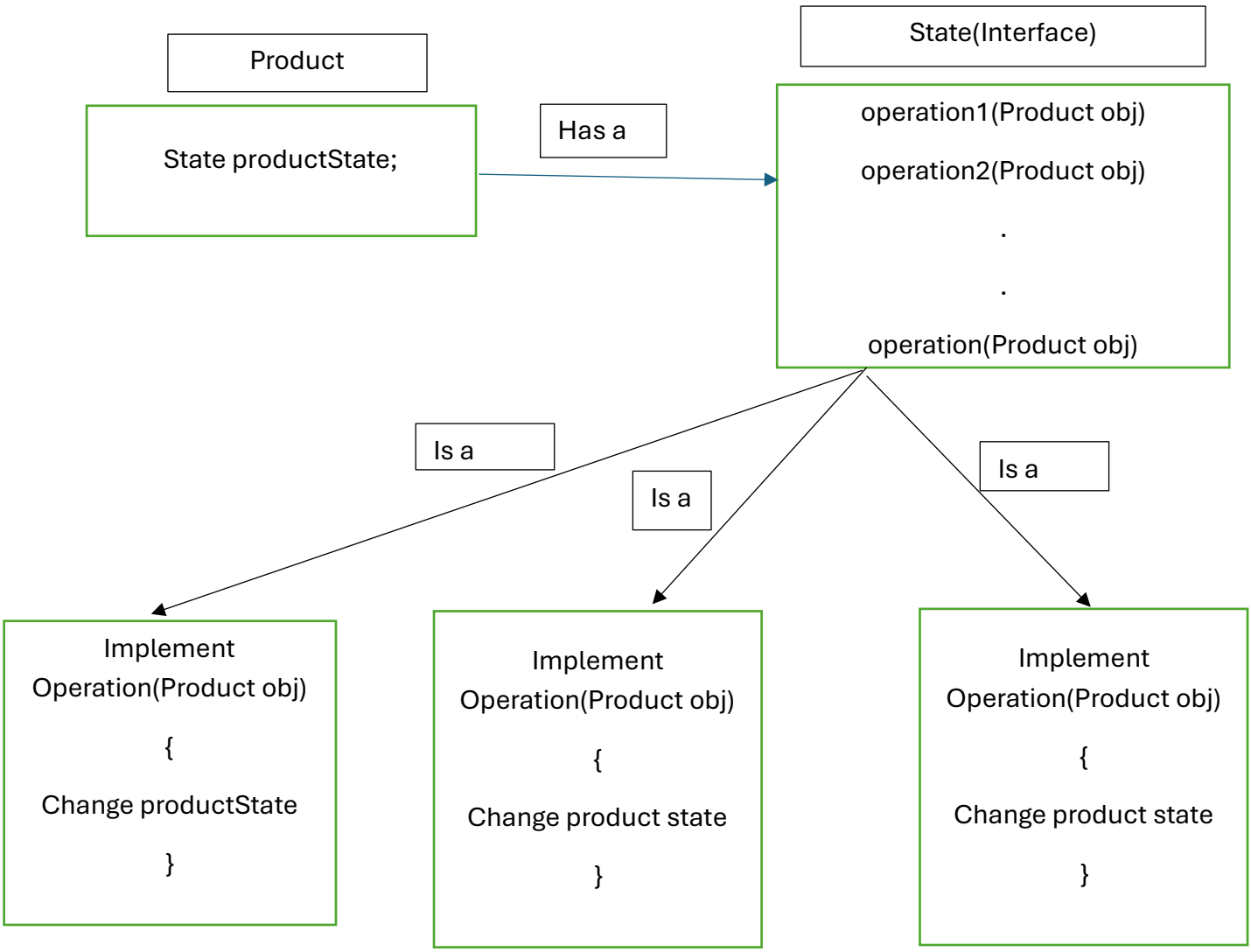
```
public interface AuctionMediator {
        void addBidder(Colleague bidder);
        void placeBid(Colleague bidder , int bidAmount);
}
```

//**Mediator Concrete Class**

```
public class Auction implements AuctionMediator {
        List<Colleague> colleagues= new ArrayList<>();
        @override
        public void addBidder(Colleague bidder) {
                colleague.add(bidder);
        }
        @override
        public void placeBid(Colleague bidder, int bidAmount) {
                for(Colleague colleague : colleagues) {
                        if(!colleague.getname().equals(bidder.getName())) {
                                colleague.receiveBidNotification(bidAmount);
                        }
                }
        }       }
```

State Design Pattern:

allows an object to alter its behavior when its internal state changes.



Vending Machine

```
public class VendingMachine {
        VendingState machineState;
        public  VendingState getMachineState() {
                return machineState;
        }
        public void setMachineState(VendingState getMachineState() {
```