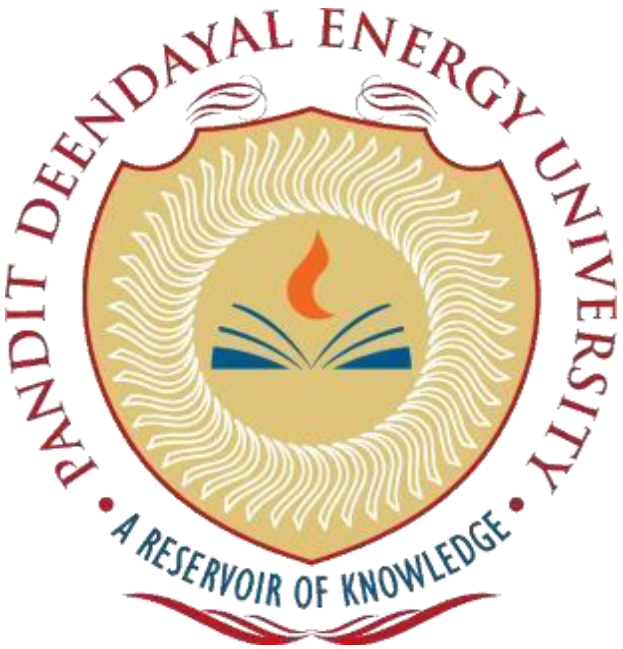
**Pandit Deendayal Energy University, Gandhinagar**

**School of Technology**

**Department of Computer Science & Engineering**

# Design Pattern Lab(20CP210P)




Name: **Padshala Smit Jagdishbhai**

Enrolment No: **21BCP187**

Semester: **IV Division: 3 (G6)**

Branch: **Computer Science Engineering**

# INDEX

# Behavioral Design Patterns: -

Behavioral design patterns are software design patterns that are concerned with the communication and interaction between objects. They are used to improve the flexibility, maintainability, and extensibility of software systems by separating the responsibilities of objects and defining the protocols of their interactions. Here's a brief overview of some of the most commonly used behavioral design patterns:

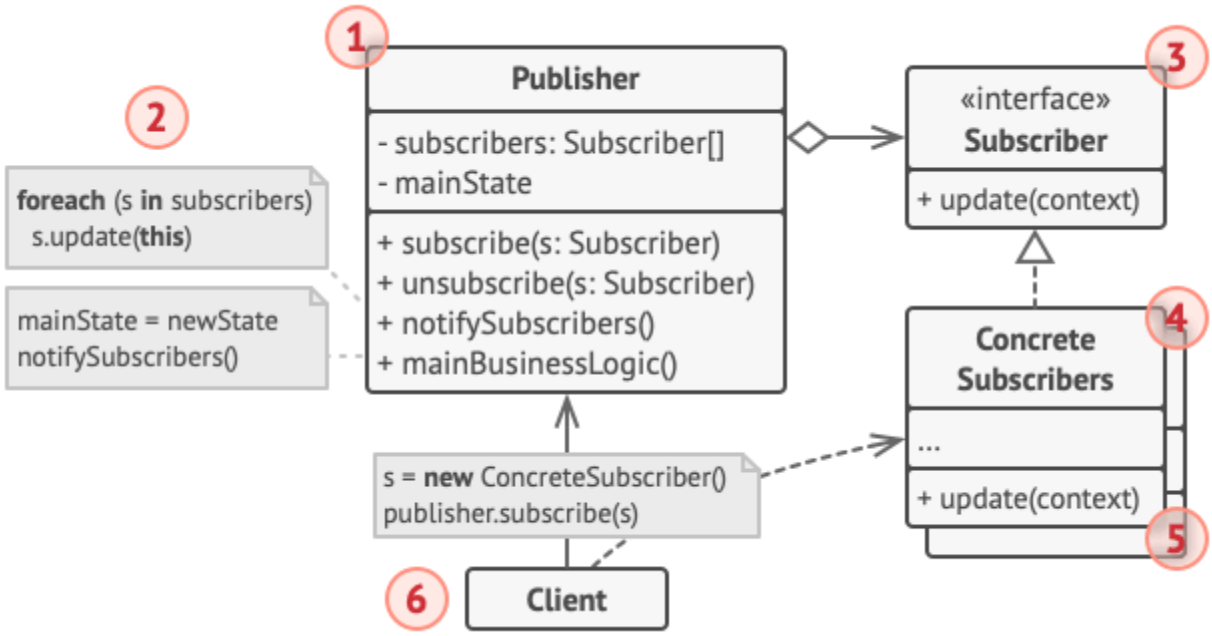
1. **Observer Pattern -** Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
2. **Memento Pattern -** Allows an object to capture its internal state and save it externally, so that the object can be restored to that state later without violating encapsulation. Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
3. **Mediator Pattern -** Defines an object that encapsulates the communication and coordination between a set of objects, so that they can interact with each other without knowing each other's identities. Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
4. **State Pattern -** Allows an object to change its behavior when its internal state changes, by encapsulating the different behaviors in separate state objects and delegating the behavior to the current state object. State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
5. **Iterator Pattern -** Provides a way to access the elements of an object sequentially without exposing its internal representation, by defining an object that can sequentially access the elements of another object. Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

# 1. Observer Behavioral Design Pattern: -

The Observer Behavioral Design Pattern is a software design pattern used in object-oriented programming. It allows an object, called the subject, to maintain a list of its dependents, called observers, and notify them automatically of any state changes. This pattern helps to decouple the subject and observer objects, promoting better code reusability and maintainability. The Observer pattern is widely used in event-driven systems and GUI frameworks, where user actions trigger changes in the system state. The Observer pattern is part of the behavioral design patterns, which focus on communication between objects and how they interact to achieve a desired behavior.

## Advantages: -

1. Provides loose coupling between objects
2. Increases flexibility and modularity of the code
3. Allows for easy maintenance and modification of the codebase




## Disadvantages: -

1. Can lead to overhead and decreased performance
2. Can result in complex implementation, especially in large systems

## Code: -

```
import java.util.*;

class SubjectEntity{
   private int numberstate;
   private List<ObserverEntity> lm = new ArrayList<ObserverEntity>();

   public void setNumberstate(int n) {
      numberstate = n;
      notifyAllObserver();
   }

   public int getNumberstate() {
      return numberstate;
   }

   public void registerObserver(ObserverEntity e) {
      lm.add(e);
   }

   public void UnregisterObserver(ObserverEntity e) {
      lm.remove(e);
   }

   public void notifyAllObserver() {
      for(ObserverEntity o : lm ){
         o.update();
      }
   }
}

abstract class ObserverEntity{
   SubjectEntity se;
   abstract public void update();
}

class BinaryObserverEntity extends ObserverEntity{

   public BinaryObserverEntity(SubjectEntity e){
      se = e;
      se.registerObserver(this);
```

```
    }
    public void update() {
        int n = se.getNumberstate();
        System.out.println("Representing number : " + n + " to binary : "+
Integer.toBinaryString(n));
    }

}
class HexaObserverEntity extends ObserverEntity{

    public HexaObserverEntity(SubjectEntity e){
        se = e;
        se.registerObserver(this);
    }
    public void update() {
        int n = se.getNumberstate();
        System.out.println("Representing number : " + n + " to binary : "+
Integer.toHexString(n));
    }

}
class OctalObserverEntity extends ObserverEntity{

    public OctalObserverEntity(SubjectEntity e){
        se = e;
        se.registerObserver(this);
    }
    public void update() {
        int n = se.getNumberstate();
        System.out.println("Representing number : " + n + " to binary : "+
Integer.toOctalString(n));
    }

}

public class ObserverDesignPattern {
    public static void main(String[] args) {
        SubjectEntity se = new SubjectEntity();
        ObserverEntity o1 = new BinaryObserverEntity(se);
        ObserverEntity o2 = new HexaObserverEntity(se);
        ObserverEntity o3 = new OctalObserverEntity(se);
```

```
        // se.registerObserver(o1);  //It is must required
        se.setNumberstate(10);
        se.UnregisterObserver(o2);
        se.setNumberstate(20);
    }
}
```

**Output: -**

```
Representing number : 10 to binary : 1010
Representing number : 10 to binary : a
Representing number : 10 to binary : 12
Representing number : 20 to binary : 10100
Representing number : 20 to binary : 24
```

# 2. Memento Behavioral Design Pattern: -

The Memento Design Pattern is a behavioral design pattern that allows you to save and restore the internal state of an object without violating its encapsulation. It provides a way to capture the current state of an object and store it externally so that the object can be restored to that state later. The Memento pattern consists of three components: the Originator, the Memento, and the Caretaker.

The Originator is the object that has the internal state that needs to be saved and restored. The Memento is an object that stores the state of the Originator. The Caretaker is an object that is responsible for storing and retrieving Mementos.

## Advantages: -

We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.

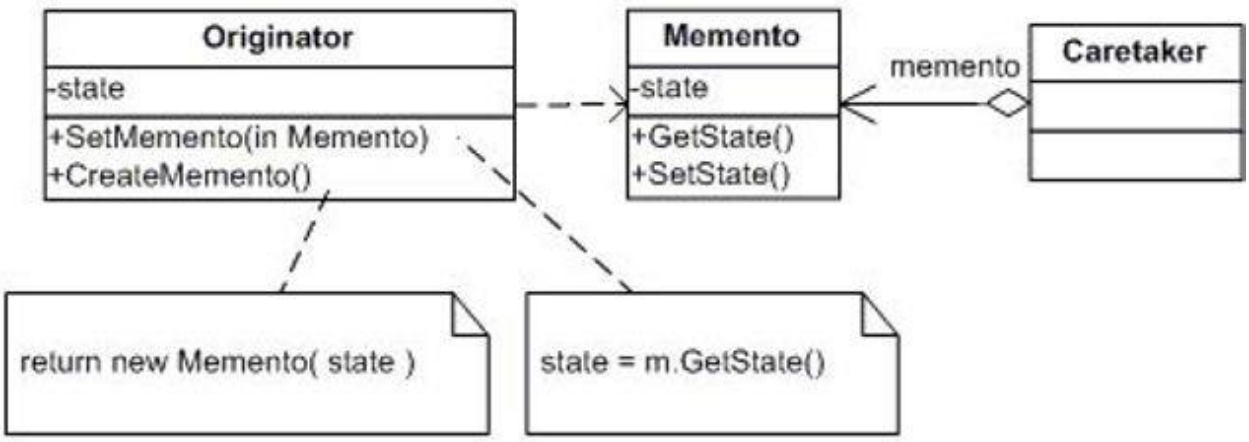
## Disadvantages: -

If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

**Use the Memento pattern when:**

A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.

A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.



## Code: -

```
import java.util.ArrayList;
import java.util.List;

class OriginatorObject{
   private int temperature;
   private int volume;

   public OriginatorObject(int t, int v){
      temperature = t;
      volume = v;
   }

   public void setTemperature(int t) {
      temperature = t;
   }
   public void setVolume(int v) {
      volume = v;
   }
   public int getTemperature() {
      return temperature;
   }
   public int getVolume() {
      return volume;
   }

   public MementoObject saveOriginatorObjectSate(){
      return new MementoObject(temperature, volume);
   }

   public void restoreOriginatorObjectState(MementoObject m){
      temperature = m.getTemperature();
      volume = m.getVolume();
   }
   public void operatorMachine(){
      System.out.println("Machine operating with temperature :"+temperature+" and
volume: "+volume);
   }
}
```

```
class MementoObject{
   private int temperature;
   private int volume;

   public MementoObject(int t, int v){
      temperature = t;
      volume = v;
   }
   public int getTemperature() {
      return temperature;
   }
   public int getVolume() {
      return volume;
   }
}

class CareTakerObject{
   List<MementoObject> lm = new ArrayList<MementoObject>();
   public void addMementoObject(MementoObject m){
      lm.add(m);
   }

   public MementoObject getMementoObject(int i){
      return lm.get(i);
   }
}

public class memento_dp{
   public static void main(String[] args) {
      OriginatorObject o1  = new OriginatorObject(20, 10);
      CareTakerObject c1 = new CareTakerObject();
      o1.operatorMachine();
      c1.addMementoObject(o1.saveOriginatorObjectSate());

      o1.setTemperature(27);
      o1.setVolume(30);
      o1.operatorMachine();
      c1.addMementoObject(o1.saveOriginatorObjectSate());

      o1.setTemperature(37);
      o1.setVolume(50);
      o1.operatorMachine();
```

```
        c1.addMementoObject(o1.saveOriginatorObjectSate());

        // Restoring the previous state
        o1.restoreOriginatorObjectState(c1.getMementoObject(0));
        o1.operatorMachine();
    }
}
```

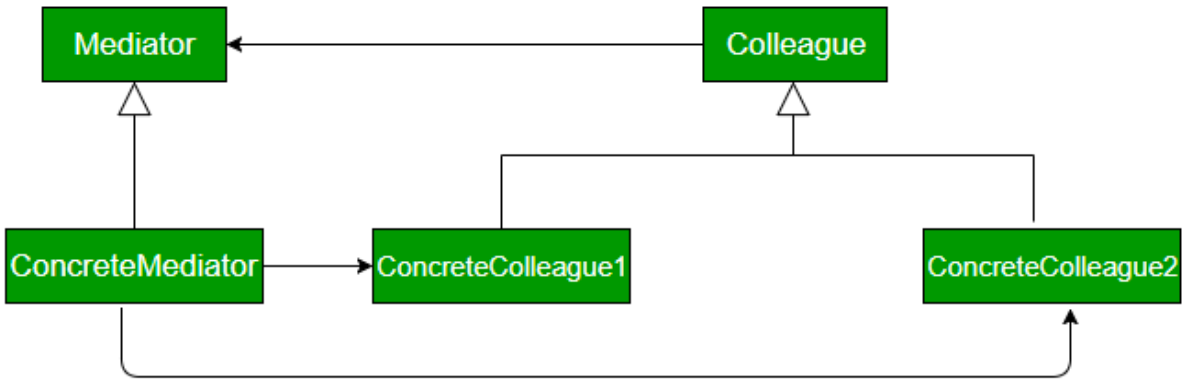
**Output: -**

```
Machine operating with temperature :20 and volume: 10
Machine operating with temperature :27 and volume: 30
Machine operating with temperature :37 and volume: 50
Machine operating with temperature :20 and volume: 10
```

# 3. Mediator Behavioral Design Pattern: -

The Mediator Design Pattern is a behavioral design pattern that allows for communication between objects without them having to directly reference each other. It promotes loose coupling by keeping objects from knowing too much about each other, instead, they communicate through a mediator.

The mediator acts as a central hub for communication between the objects it manages. When an object needs to communicate with another object, it sends a message to the mediator, which then relays the message to the appropriate object. This reduces the number of connections between objects and makes it easier to add, remove, or modify objects without affecting the rest of the syst




**Design components**

**Mediator:** It defines the interface for communication between colleague objects.

**Concrete Mediator:** It implements the mediator interface and coordinates communication between colleague objects.

**Colleague:** It defines the interface for communication with other colleagues

**Concrete Colleague:** It implements the colleague interface and communicates with other colleagues through its mediator

**Advantages: -**

1. Decouples objects by removing direct references to each other.
2. Makes it easy to add, remove, or modify objects and adapt to changing requirements or scale the system.
3. Simplifies communication between objects by providing a central hub.

**Disadvantages: -**

1. Adds complexity to the system by introducing a new layer of abstraction.
2. Can have a performance impact on the system if the mediator becomes a bottleneck.

## Code: -

```
import java.util.*;
// Mediator Interface
interface ArmyMediator {
    public void send(String message, ArmyTeam sender);
}
// Collegue Class - Army Team
class ArmyTeam {
    private String teamName;
    private ArmyMediator mediator;
    public ArmyTeam(String teamName, ArmyMediator mediator) {
        this.teamName = teamName;
        this.mediator = mediator;
    }
    public void send(String message) {
        System.out.println("[" + teamName + "] Sending message: " + message);
        mediator.send(message, this);
    }
    public void receive(String message) {
        System.out.println("[" + teamName + "] Received message: " + message);
    }
}

// Concrete Mediator Class - Army Commander
class ArmyCommander implements ArmyMediator {
    private List<ArmyTeam> teams = new ArrayList<>();
    public void addTeam(ArmyTeam team) {
        teams.add(team);
    }
    public void send(String message, ArmyTeam sender) {
        for (ArmyTeam team : teams) {
            if (team != sender) {
                team.receive(message);
```

```
            }
        }
        System.out.println();
    }
}
public class ArmyMediatorExample {
    public static void main(String[] args) {
        // Create mediator (Army Commander)
        ArmyCommander commander = new ArmyCommander();
        // Create army teams
        ArmyTeam team1 = new ArmyTeam("Team1", commander);
        ArmyTeam team2 = new ArmyTeam("Team2", commander);
        ArmyTeam team3 = new ArmyTeam("Team3", commander);
        // Add teams to the mediator (Army Commander)
        commander.addTeam(team1);
        commander.addTeam(team2);
        commander.addTeam(team3);
        // Send messages between army teams
        team1.send("Attack the enemy from the east!");
        team2.send("Hold the defensive line!");
        team3.send("Provide support to team1!");
    }
}
```

**Output: -**

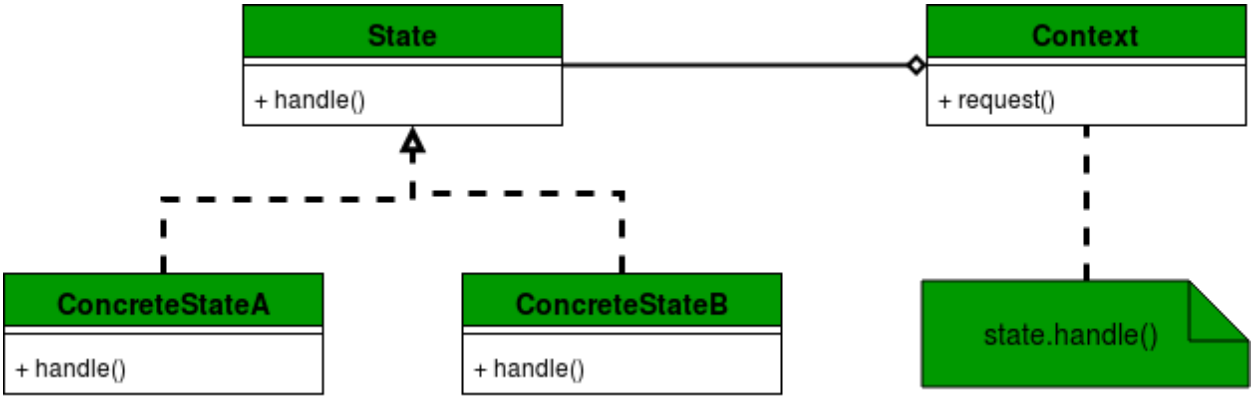
```
[Team1] Sending message: Attack the enemy from the east!
[Team2] Received message: Attack the enemy from the east!
[Team3] Received message: Attack the enemy from the east!

[Team2] Sending message: Hold the defensive line!
[Team1] Received message: Hold the defensive line!
[Team3] Received message: Hold the defensive line!

[Team3] Sending message: Provide support to team1!
[Team1] Received message: Provide support to team1!
[Team2] Received message: Provide support to team1!
```

# 4. State Behavioral Design Pattern: -

The state pattern is one of the behavioral design patterns. A state design pattern is used when an Object changes its behavior based on its internal state. If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use the if-else condition block to perform different actions based on the state. The state pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.




**Context:** Defines an interface for clients to interact. It maintains references to concrete state objects which may be used to define the current state of objects.

**State:** Defines interface for declaring what each concrete state should do.

**Concrete State:** Provides the implementation for methods defined in State.

## Advantages: -

1. Encapsulates states and transitions between states.
2. Simplifies code by reducing conditional statements and improving readability.
3. Increases extensibility and maintainability by separating state-specific behavior from the main logic.

## Disadvantages: -

1. Can increase the complexity of the system by introducing new classes and interfaces.
2. Can reduce performance by adding additional overhead for state transitions.

## Code: -

```
interface State {
   public void handleState();
}

class State1 implements State{
   public void handleState(){
      System.out.println("Handling State1 - you may go to State2");
   }
}
class State2 implements State{
   public void handleState(){
      System.out.println("Handling State2 - you may go to State3");
   }
}
class State3 implements State{
   public void handleState(){
      System.out.println("Handling State3 - you may go to initial State!");
   }
}

class Context{
   State st;
   public Context(State s){
      st=s;
   }
   public void setSt(State st) {
      this.st = st;
   }
   public State getSt() {
      return st;
   }

   public void perfomAction(){
      st.handleState();
   }
}


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

```
        State s1 = new State1();
        State s2 = new State2();
        State s3 = new State3();

        Context c1 = new Context(s2);
        c1.perfomAction();

        c1.setSt(s3);
        c1.perfomAction();
    }
}
```

**Output: -**

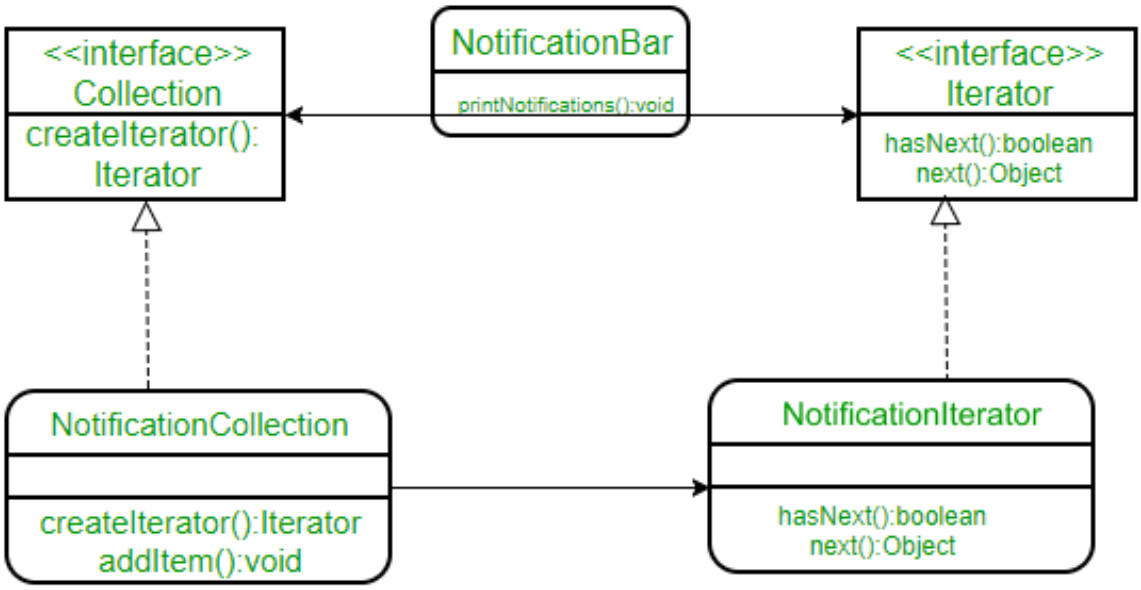
```
Handling State2 - you may go to State3
Handling State3 - you may go to initial State!
```

# 5. Iterator Behavioral Design Pattern: -

The Iterator Design Pattern is a behavioral design pattern that provides a way to access elements of a collection without exposing its underlying representation. It defines an object called an iterator that is responsible for iterating over the elements of a collection.

The iterator pattern decouples the algorithm that accesses the elements from the collection itself. This allows for the collection to be modified without affecting the algorithm that iterates over it. The iterator provides a consistent interface for accessing elements regardless of the underlying implementation of the collection.

The iterator pattern consists of two main components: the Iterator and the Aggregate. The Iterator defines the interface for accessing elements of a collection and provides methods to move to the next element and retrieve the current element. The Aggregate defines the interface for creating the Iterator object and provides a way to access the elements of the collection.




## Advantages: -

1. Encapsulates the iteration logic and provides a consistent interface for accessing elements of a collection.
2. Allows for the traversal of complex data structures without exposing their underlying implementation.
3. Simplifies code and promotes modular design.

## Disadvantages: -

1. Can increase the complexity of the system by introducing additional classes and interfaces.
2. May not be suitable for small or simple collections.
3. Performance impact if the iterator needs to traverse a large data structure or if the iterator implementation is not optimized.

## Code: -

```
import java.util.*;   // best example of this pattern is Iterator in java library
class Channel {
   private double frequency;
   private String type;
   public Channel(double frequency, String type) {
      this.frequency = frequency;
      this.type = type;
   }
   public double getFrequency() {
      return frequency;
   }
   public void setFrequency(double frequency) {
      this.frequency = frequency;
   }
   public String getType() {
      return type;
   }
   public void setType(String type) {
      this.type = type;
   }
   public String toString() {
      return "Channel [frequency=" + frequency + ", type=" + type + "]";
   }
}
class ChannelIterator {
   LinkedList<Channel> collection = new LinkedList<Channel>();
   int currentIndex=0;
   public ChannelIterator(LinkedList<Channel> collection) {
      this.collection = collection;
   }
   public Channel getNext(){
```

```
        if(hasNext()){
            return collection.get(currentIndex++);
        }
        return null;
    }  //til hasNext is true return Channel
    public boolean hasNext(){
        return currentIndex < collection.size();
    }
}
class ChannelCollection {
    LinkedList<Channel> collection = new LinkedList<Channel>();
    public void addChannel(Channel c){
        collection.add(c);
    }
    public void removeChannel(Channel c){
        collection.remove(c);
    }
    public ChannelIterator getIterator(){
        ChannelIterator iterator = new ChannelIterator(collection);
        return iterator;
    }
}
public class IteratorDesignPatternDemo {
    public static void main(String[] args) {
        ChannelCollection cc = new ChannelCollection();
        cc.addChannel(new Channel(98.5, "Radio Mirchi"));
        cc.addChannel(new Channel(93.5, "RedFM"));
        cc.addChannel(new Channel(104.0, "Hangama Radio"));
        ChannelIterator iterator = cc.getIterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.getNext());
        }
    }
}
```

**Output: -**

```
Channel [frequency=98.5, type=Radio Mirchi]
Channel [frequency=93.5, type=RedFM]
Channel [frequency=104.0, type=Hangama Radio]
```