Experiment Title: Behavioral Design Pattern I (Chain of Responsibility Pattern & Iterator Pattern)

Aim/Objective: To analyse the implementation of Chain of Responsibility Design Pattern & Iterator Design Patterns for the real-time scenario.

Description:

The student will understand the concept of Chain of Responsibility Design Pattern & Iterator Design Patterns.

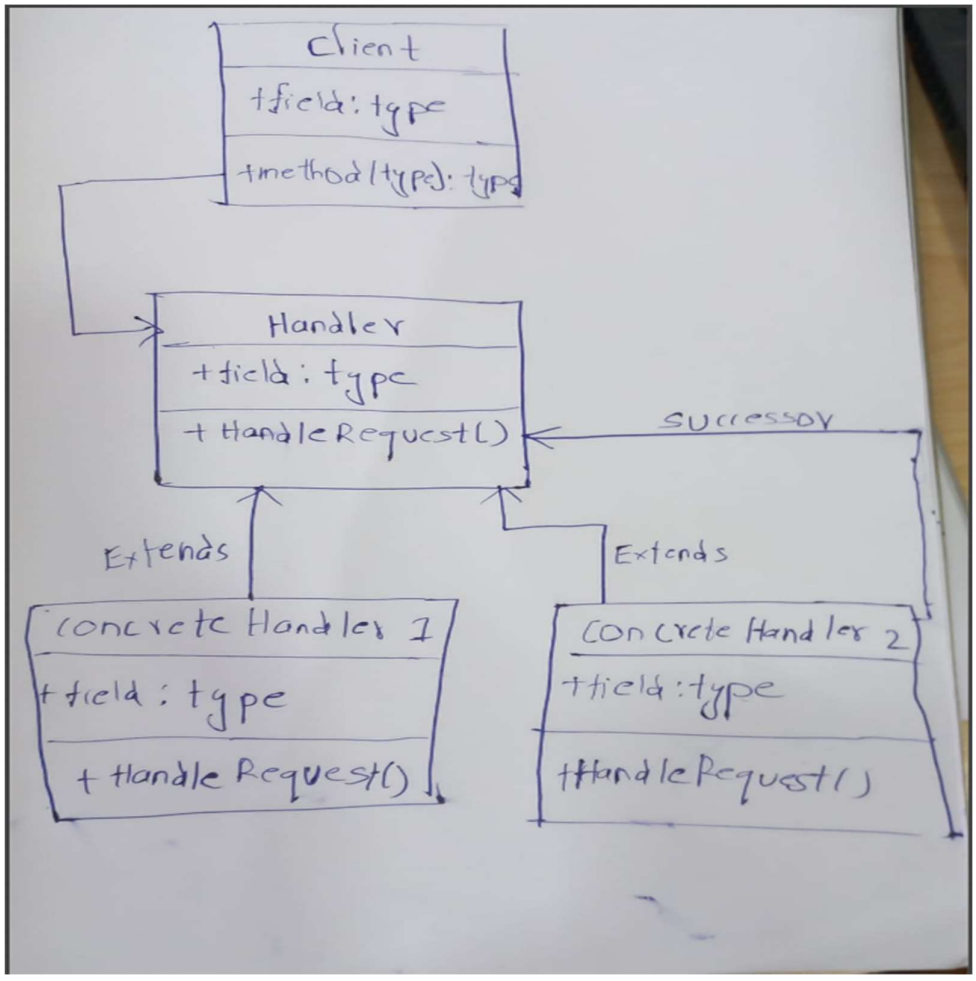
Pre-Requisites:

Knowledge: Classes and Objects in JAVA

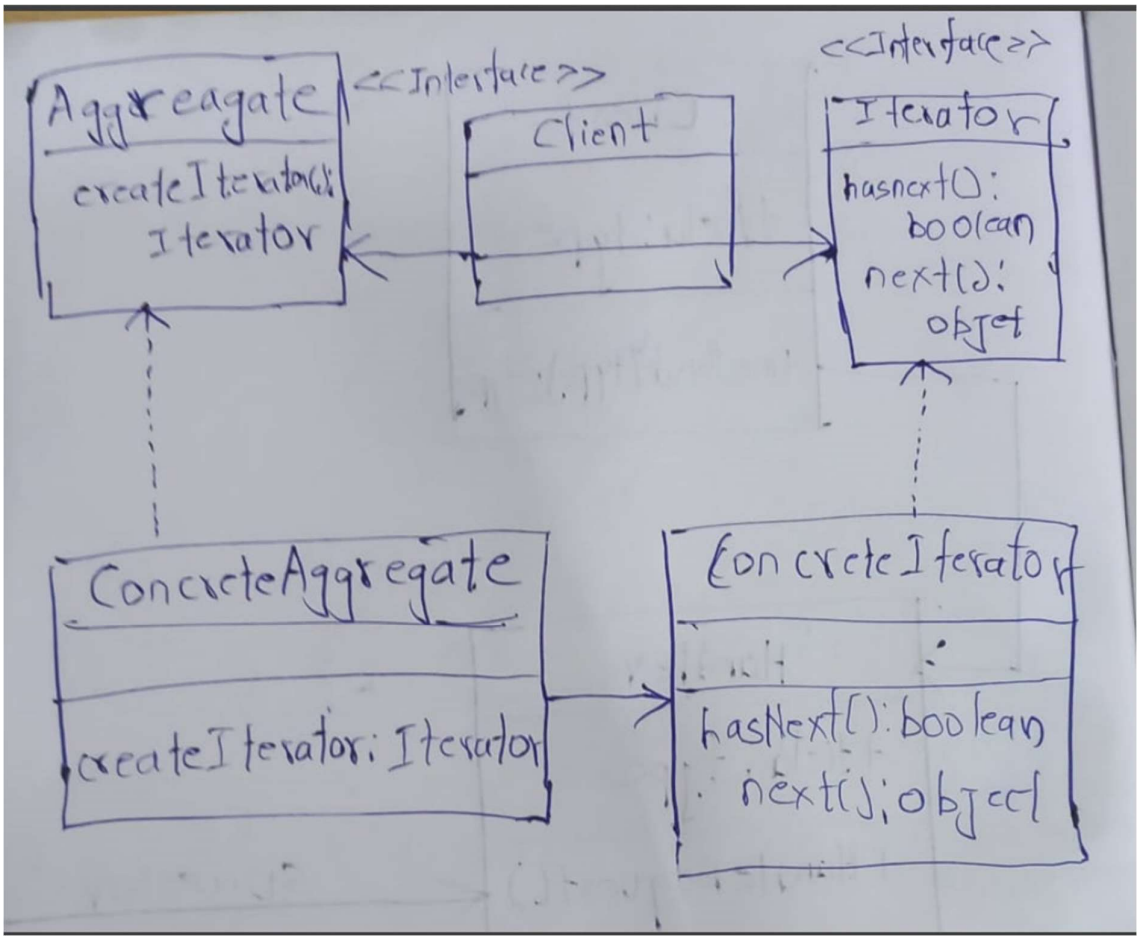
Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1. Draw the UML Relationship Diagram for Chain of Responsibility Design Pattern for customized Scenarios.

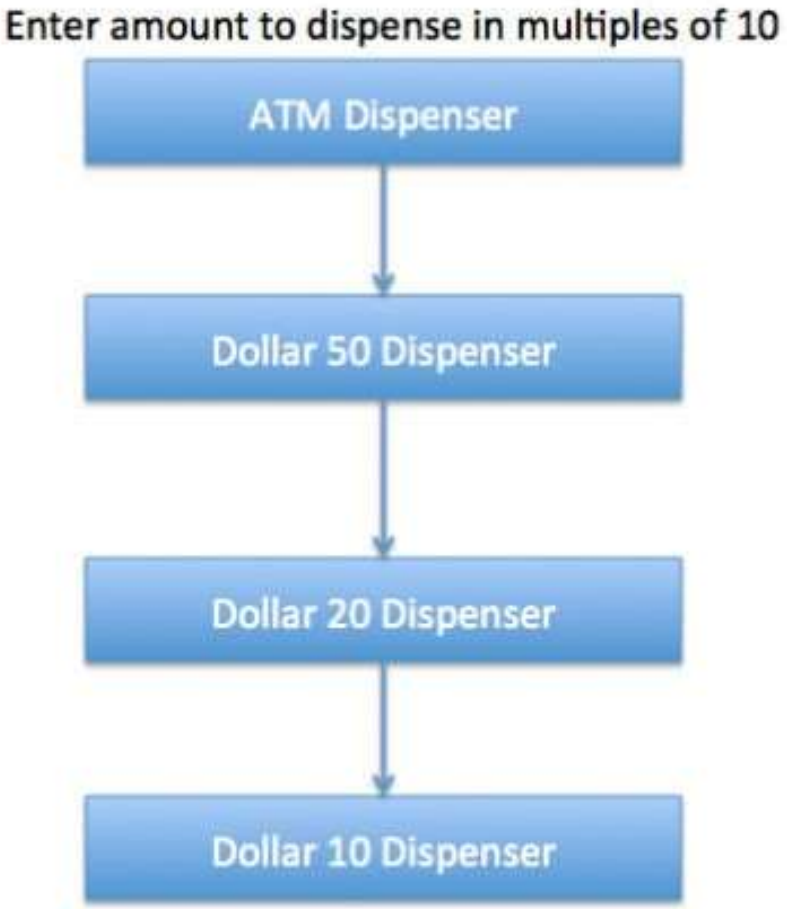


2. Draw the UML Relationship Diagram for Iterator Design` Pattern for customized Scenarios.



In-Lab:

Implement the Chain of Responsibility pattern in **ATM Dispense machine Scenario**.
The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50$, 20$, 10$ etc. If the user enters an amount that is not multiples of 10, it throws error. We will use Chain of Responsibility pattern to implement this solution. The chain will process the request in the same order as below image.



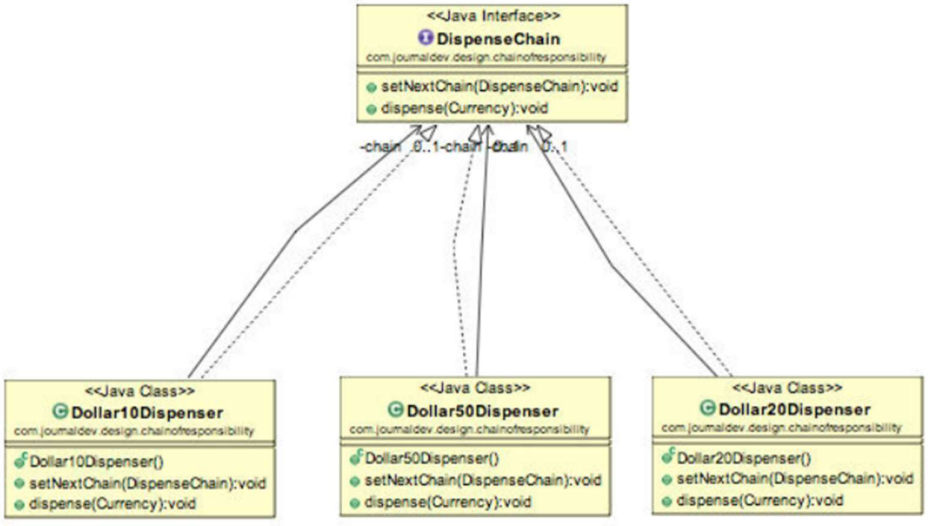
The implementation this in a single program will increase the complexity and the solution will be tightly coupled.

So, It has to be implemented as chain of dispense systems to dispense bills of 50$, 20$ and 10$.

```
// 1. DispenseChain Interface
interface DispenseChain {
    void setNextChain(DispenseChain nextChain);
    void dispense(int amount);
}

// 2. DispenseHandler Abstract Class
abstract class DispenseHandler implements DispenseChain {
    protected DispenseChain nextChain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.nextChain = nextChain;
    }
}

// 3. BillDispenser Class - Concrete Handler
class BillDispenser extends DispenseHandler {
    private int denomination;

    public BillDispenser(int denomination) {
        this.denomination = denomination;
    }

    @Override
    public void dispense(int amount) {
        int numBills = amount / denomination;
        int remainingAmount = amount % denomination;
        if (numBills > 0) {
            System.out.println("Dispensing " + numBills + " $" + denomination + " bill(s)");
        }
        if (remainingAmount > 0 && nextChain != null) {
            nextChain.dispense(remainingAmount);
        }
    }
}

// 4. ATMDispenser Class - Client
public class ATMDispenser {
    private DispenseChain dispenserChain;

    public ATMDispenser() {
        buildDispenserChain();
    }
```

```
    private void buildDispenserChain() {
        DispenseChain bill50Dispenser = new BillDispenser(50);
        DispenseChain bill20Dispenser = new BillDispenser(20);
        DispenseChain bill10Dispenser = new BillDispenser(10);

        bill50Dispenser.setNextChain(bill20Dispenser);
        bill20Dispenser.setNextChain(bill10Dispenser);

        dispenserChain = bill50Dispenser;
    }

    public void dispense(int amount) {
        if (amount % 10 != 0) {
            System.out.println("Amount should be multiples of 10.");
            return;
        }

        dispenserChain.dispense(amount);
    }

    public static void main(String[] args) {
        ATMDispenser atmDispenser = new ATMDispenser();
        atmDispenser.dispense(170);
    }
}
```

Refer the below mentioned Structure.



```
 package com.journaldev.design.chainofresponsibility;
public class Currency
{
private int amount;
public Currency(int amt){
 this.amount=amt;
}
public int getAmount()
{
return this.amount;
}
}


package com.journaldev.design.chainofresponsibility;
public interface DispenseChain
 {
void setNextChain (DispenseChain nextChain);
 }
void dispense (Currency cur);


 public class Dollar50Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) { this.chain-nextChain;

}

@Override

public void dispense (Currency cur) { if(remainder !=0) this.chain.dispense (new Currency(remainder));

if (cur.getAmount() >= 50){

int num = cur.getAmount()/50; int remainder = cur.getAmount() % 50;

System.out.println("Dispensing "+num+" 50$ note");

}else{

this.chain.dispense (cur);

}

}

}

package com.journaldev.design.chainofresponsibility;

public class Dollar20Dispenser implements DispenseChain{

private DispenseChain chain;
```

```
@Override
public void setNextChain (DispenseChain nextChain) { this.chain=nextChain;
}
@Override
if (cur.getAmount() >= 20){
public void dispense (Currency cur) { int remainder = cur.getAmount() % 20;
int num = cur.getAmount()/20;
System.out.println("Dispensing "+num+" 20$ note");
if(remainder !=0) this.chain. dispense (new Currency (remainder));
}else{
this.chain.dispense (cur);
}
}
}
package com.journaldev.design.chainofresponsibility;
public class Dollar10Dispenser implements DispenseChain { private DispenseChain chain;
@Override
public void setNextChain (DispenseChain nextChain) { this.chain-nextChain;
}
@Override
public void dispense (Currency cur) { if (cur.getAmount() >= 10){
int num = cur.getAmount()/10;
int remainder = cur.getAmount() % 10;
System.out.println("Dispensing "+num+" 10$ note");
if(remainder !=0) this.chain.dispense (new Currency(remainder));
}else{
this.chain.dispense (cur);
}
}
}
```

```
package com.journaldev.design.chainofresponsibility;

import java.util.Scanner;

public class ATMDispenseChain {

private DispenseChain c1;

public ATMDispenseChain() {

// initialize the chain

this.c1 = new Dollar50Dispenser();

DispenseChain c2 = new Dollar20Dispenser(); DispenseChain c3 = new Dollar10Dispenser();

// set the chain of responsibility c1.setNextChain(c2);

c2.setNextChain (c3);

}

public static void main(String[] args) { ATMDispenseChain atmDispenser = new ATMDispenseChain(); while (true) {

int amount = 0;

System.out.println("Enter amount to dispense");

Scanner input = new Scanner(System.in);

amount = input.nextInt();

if (amount % 10 != 0) { System.out.println("Amount should be in multiple of 10s."); return;

} // process the request atmDispenser.c1.dispense (new Currency (amount));

}

}

}
```

Sample VIVA-VOCE Questions (In-Lab):

1. **State at which situation that we need Chain of Responsibility Design Pattern.**

Where and When Chain of Responsibility pattern is applicable : When you want to decouple a request's sender and receiver. Multiple objects, determined at runtime, are candidates to handle a request. When you don't want to specify handlers explicitly in your code

2. **Discuss about Chain of Responsibility Design Pattern by considering the scenario of an IT Industry Organization of Roles on various Designations**

Scenario Description:

Imagine an IT company with multiple departments and various roles within each department. When an employee has a query or needs approval for something, the request should be handled by the relevant authority based on the designation hierarchy.

Designation Hierarchy:

In our IT company, we have the following designation hierarchy:

Junior Developer

Senior Developer

Team Lead

Project Manager

Department Head

CEO

Chain of Responsibility Implementation:

To implement the Chain of Responsibility pattern, we create a base handler interface or abstract class that defines the common methods and a reference to the next handler in the chain. Each role in the designation hierarchy will have its concrete handler class that implements the base handler interface and contains the logic to handle the request or pass it to the next handler in the chain.

3. **Illustrate the difference between Chain of Responsibility and Iterator design pattern.**

Purpose: The Chain of Responsibility pattern is used to achieve loose coupling in a scenario where a request needs to be processed by one of multiple objects, but the sender doesn't need to know which object will handle the request.

Problem: It addresses the "responsibility delegation" problem, where a request may need to pass through a chain of processing objects until it finds the appropriate handler.

Participants: The pattern involves a chain of handler objects, each having a reference to the next handler in the chain.

Interaction: When a request is made, it is passed down the chain. Each handler decides whether to handle the request or pass it to the next handler in the chain.

Typical Use Cases: Logging systems, event handling, request processing pipelines, and middleware

4. **Discuss the Pros and Cons of Iterator Design Pattern.**

Pros:

Encapsulation of Iteration Logic: The Iterator pattern encapsulates the iteration logic within the Iterator object, abstracting away the details of the underlying data structure. This helps in separating the concerns of data traversal and data structure representation.

Uniform Interface: The pattern provides a uniform interface to access elements in different types of collections. It allows clients to access elements sequentially without worrying about the specific implementation details of the collection.

Cons:

Increased Complexity: Implementing the Iterator pattern can introduce some level of complexity, especially if the collection has a complex structure or requires custom iteration logic. This could lead to additional development and maintenance effort.

Performance Overhead: Depending on the specific implementation of the Iterator, there may be a slight performance overhead compared to directly accessing elements in the collection, as it involves an additional layer of abstraction.

5. **Discuss the Pros and Cons of Chain of Responsibility Design Pattern.**

Pros:

Loose Coupling: The Chain of Responsibility pattern promotes loose coupling between the sender of a request and its potential receivers. The sender doesn't need to know which handler will process the request, and the handlers don't have direct knowledge of each other, leading to a more flexible and maintainable codebase.

Responsibility Delegation: The pattern allows the responsibility for handling a request to be delegated through a chain of handlers. Each handler has the freedom to process the request or pass it to the next handler in the chain, allowing for dynamic configuration and modification of the chain at runtime.

Cons:

Guaranteed Handling: There's no guarantee that a request will be handled at all. If the request passes through the entire chain without finding a suitable handler, it may go unhandled, leading to potential issues.

Performance Overhead: The Chain of Responsibility pattern introduces a performance overhead since each request has to traverse the entire chain before either being handled or deemed unhandled. This can impact the response time for time-sensitive operations.

Post-Lab:

Consider a scenario that admin broadcaster has a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel.

For example, some client programs are only interested in English channels and want to process only them, they don't want to process other types of channels.

So, the admin can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client must come up with the logic for traversal. It can't be assured that client logic is correct.

Furthermore, if the number of clients grows then it will become very hard to maintain.

Here as a programmer, write a Java Program to use the Iterator pattern and provide iteration based on type of channel. you should make sure that client program can access the list of channels only through the iterator.

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

class Channel {
   private String name;
   private String type;

   public Channel(String name, String type) {
      this.name = name;
      this.type = type;
   }

   public String getName() {
      return name;
   }

   public String getType() {
      return type;
   }
}

interface ChannelIterator {
   boolean hasNext();
   Channel next();
}

class AllChannelsIterator implements ChannelIterator {
```

```
    private List<Channel> channels;
    private int position;

    public AllChannelsIterator(List<Channel> channels) {
        this.channels = channels;
        this.position = 0;
    }

    @Override
    public boolean hasNext() {
        return position < channels.size();
    }

    @Override
    public Channel next() {
        if (hasNext()) {
            return channels.get(position++);
        }
        throw new IndexOutOfBoundsException("No more channels in the collection.");
    }
}

class EnglishChannelsIterator implements ChannelIterator {
    private List<Channel> channels;
    private int position;

    public EnglishChannelsIterator(List<Channel> channels) {
        this.channels = channels;
        this.position = 0;
    }

    @Override
    public boolean hasNext() {
        while (position < channels.size()) {
            if (channels.get(position).getType().equals("English")) {
                return true;
            }
            position++;
        }
        return false;
    }
    public Channel next() {
        if (hasNext()) {
            return channels.get(position++);
        }
        throw new IndexOutOfBoundsException("No more English channels in the collection.");
```

```
    }
}

class AdminBroadcaster {
    private List<Channel> channels;

    public AdminBroadcaster() {
        channels = new ArrayList<>();
    }

    public void addChannel(Channel channel) {
        channels.add(channel);
    }

    public ChannelIterator getAllChannelsIterator() {
        return new AllChannelsIterator(channels);
    }

    public ChannelIterator getEnglishChannelsIterator() {
        return new EnglishChannelsIterator(channels);
    }
}

public class RadioClient {
    public static void main(String[] args) {
        AdminBroadcaster admin = new AdminBroadcaster();
        admin.addChannel(new Channel("Channel 1", "English"));
        admin.addChannel(new Channel("Channel 2", "Hindi"));
        admin.addChannel(new Channel("Channel 3", "English"));
        admin.addChannel(new Channel("Channel 4", "Spanish"));

        ChannelIterator allIterator = admin.getAllChannelsIterator();
        while (allIterator.hasNext()) {
            Channel channel = allIterator.next();
            System.out.println("Channel Name: " + channel.getName() + ", Type: " + channel.getType());
        }

        System.out.println("---");

        ChannelIterator englishIterator = admin.getEnglishChannelsIterator();
        while (englishIterator.hasNext()) {
            Channel channel = englishIterator.next();
   System.out.println("Channel Name: " + channel.getName() + ", Type: " + channel.getType());
        }
```

Evaluator Remark (if Any):	**Marks Secured: _____out of 50**
	Signature of the Evaluator with Date