**Chain of Responsibility Design Pattern**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/chain-of-responsibility-design-pattern.jpg)

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

**Chain of Responsibility Pattern Example in JDK**

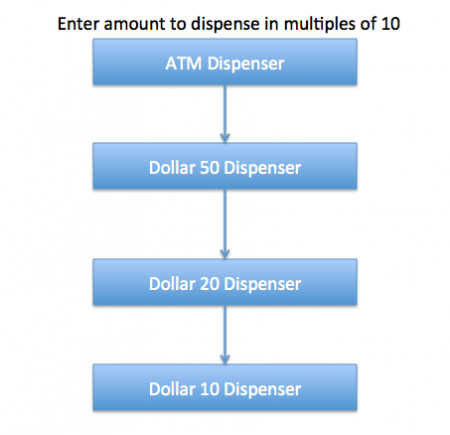
Let’s see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern. We know that we can have multiple catch blocks in a [try-catch block](https://www.journaldev.com/592/java-try-with-resources) code. Here every catch block is kind of a processor to process that particular exception.

So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

**Chain of Responsibility Design Pattern Example**

One of the great example of Chain of Responsibility pattern is **ATM Dispense machine**. 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.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Chain-of-Responsibility-Pattern.png)

Note that we can implement this solution easily in a single program itself but then the complexity will increase and the solution will be tightly coupled. So we will create a chain of dispense systems to dispense bills of 50$, 20$ and 10$.

**Chain of Responsibility Design Pattern – Base Classes and Interface**

We can create a class Currency that will store the amount to dispense and used by the chain implementations.

Currency.java

public class Currency {

private int amount;

public Currency(int amt){

this.amount=amt;

}

public int getAmount(){

return this.amount;

}

}

The base interface should have a method to define the next processor in the chain and the method that will process the request. Our ATM Dispense interface will look like below.

DispenseChain.java

public interface DispenseChain {

void setNextChain(DispenseChain nextChain);

void dispense(Currency cur);

}

**Chain of Responsibilities Pattern – Chain Implementations**

We need to create different processor classes that will implement the DispenseChain interface and provide implementation of dispense methods. Since we are developing our system to work with three types of currency bills – 50$, 20$ and 10$, we will create three concrete implementations.

Dollar50Dispenser.java

public class Dollar50Dispenser implements DispenseChain {

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain=nextChain;

}

@Override

public void dispense(Currency cur) {

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

int num = cur.getAmount()/50;

int remainder = cur.getAmount() % 50;

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

if(remainder !=0) this.chain.dispense(new Currency(remainder));

}else{

this.chain.dispense(cur);

}

}

}

Dollar20Dispenser.java

public class Dollar20Dispenser implements DispenseChain{

private DispenseChain chain;

@Override

public void setNextChain(DispenseChain nextChain) {

this.chain=nextChain;

}

@Override

public void dispense(Currency cur) {

if(cur.getAmount() >= 20){

int num = cur.getAmount()/20;

int remainder = cur.getAmount() % 20;

System.out.println("Dispensing "+num+" 20$ note");

if(remainder !=0) this.chain.dispense(new Currency(remainder));

}else{

this.chain.dispense(cur);

}

}

}

Dollar10Dispenser.java

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);

}

}

}

The important point to note here is the implementation of dispense method. You will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it.

If one of the chain not able to process it fully, it sends the request to the next processor in chain to process the remaining request. If the processor is not able to process anything, it just forwards the same request to the next chain.

**Chain of Responsibilities Design Pattern – Creating the Chain**

This is a very important step and we should create the chain carefully, otherwise a processor might not be getting any request at all. For example, in our implementation if we keep the first processor chain as Dollar10Dispenser and then Dollar20Dispenser, then the request will never be forwarded to the second processor and the chain will become useless.

Here is our ATM Dispenser implementation to process the user requested amount.

ATMDispenseChain.java

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));

}

}

}

When we run above application, we get output like below.

Enter amount to dispense

530

Dispensing 10 50$ note

Dispensing 1 20$ note

Dispensing 1 10$ note

Enter amount to dispense

100

Dispensing 2 50$ note

Enter amount to dispense

120

Dispensing 2 50$ note

Dispensing 1 20$ note

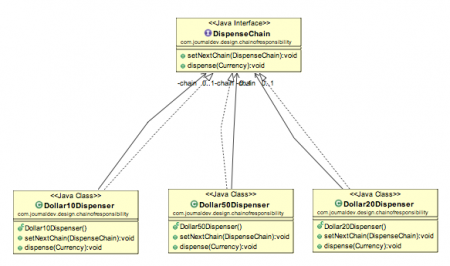
Enter amount to dispense

15

Amount should be in multiple of 10s.

**Chain of Responsibilities Design Pattern Class Diagram**

Our ATM dispense example of chain of responsibility design pattern implementation looks like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Chain-of-Responsibility-Class-Diagram.png)

**Chain of Responsibility Design Pattern Important Points**

* Client doesn’t know which part of the chain will be processing the request and it will send the request to the first object in the chain. For example, in our program ATMDispenseChain is unaware of who is processing the request to dispense the entered amount.
* Each object in the chain will have it’s own implementation to process the request, either full or partial or to send it to the next object in the chain.
* Every object in the chain should have reference to the next object in chain to forward the request to, its achieved by [java composition](https://www.journaldev.com/1325/composition-in-java-example).
* Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request. In my implementation, I have added the check for the user entered amount to make sure it gets processed fully by all the processors but we might not check it and throw exception if the request reaches the last object and there are no further objects in the chain to forward the request to. This is a design decision.
* Chain of Responsibility design pattern is good to achieve lose coupling but it comes with the trade-off of having a lot of implementation classes and maintenance problems if most of the code is common in all the implementations.

**Chain of Responsibility Pattern Examples in JDK**

* java.util.logging.Logger#log()
* javax.servlet.Filter#doFilter()

Thats all for the Chain of Responsibility design pattern, I hope you liked it and its able to clear your understanding on this design pattern.