Chain of Responsibility Design Pattern in Java

APRIL 4, 2018 BY PANKAJ — 33 COMMENTS

Chain of responsibility design pattern is one of the **behavioral design pattern**.

Table of Contents [hide]

Chain of Responsibility Design Pattern

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

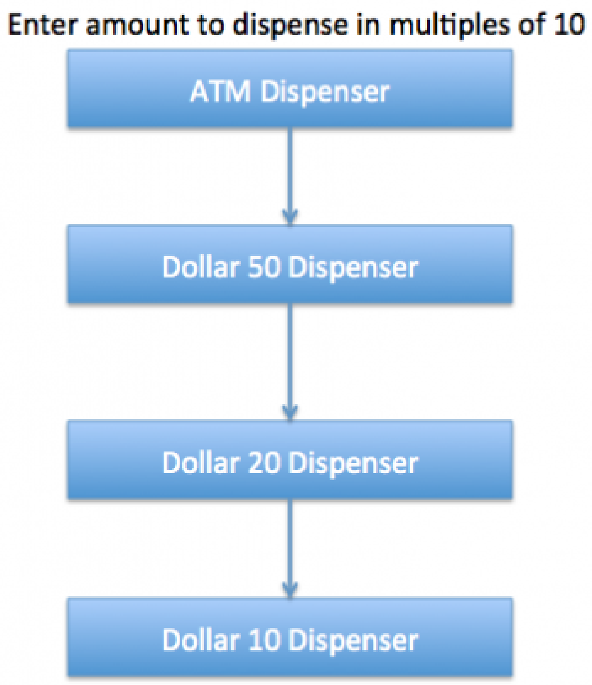
Let's see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern. We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception.

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.



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

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

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

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

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

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

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

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

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

```
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);
                }
```

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`

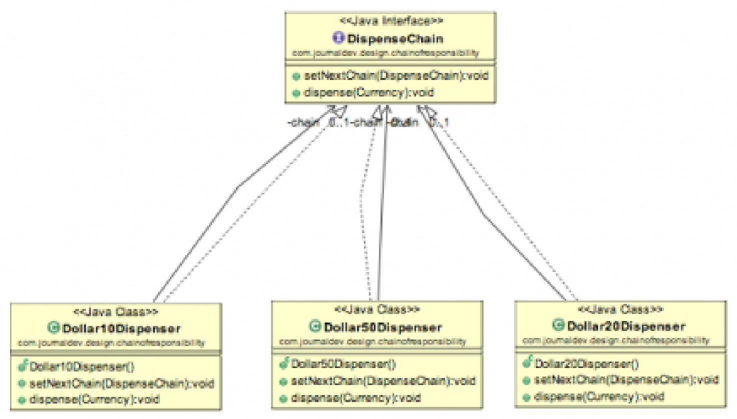
```
package com.journaldev.design.chainofresponsibility;

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();
```

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.



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.
- 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.

« PREVIOUS
Proxy Design Pattern

NEXT »
Command Design Pattern

About Pankaj
If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

isivroes says
JULY 13, 2018 AT 6:11 PM
Nice example but!!! I don´t undert why in main method do you have a while(true).
Reply

Tom says
JULY 26, 2018 AT 7:49 AM
No particular reason. It's just so you can try as many examples as you want without having to restart the program every time It's not necessary for the design pattern.
Reply

Ramazan says
JULY 3, 2018 AT 2:09 AM
Great explanations and sample!
Thank you bro.
Reply

Rajeev says
JUNE 25, 2018 AT 6:59 AM

Great blog and perfect explanation with exampl.
Thanks
Reply

现超 says
JUNE 7, 2018 AT 8:52 PM
public class Dollar10Dispenser implements DispenseChain
this.chain.dispense(cur);
nullPointException occur
Reply

Prasanti says
MARCH 27, 2018 AT 10:17 AM
Nice example and well explained.
Reply

Abdul Qadir says
JANUARY 21, 2018 AT 1:35 PM
In the Dispence method of Dollar50Dispenser, Dollar20Dispenser and Dollar10Dispenser, else part is not required i guess. Request is already getting processed to the next chain dispense if remainder != 0 .
Correct me if i am wrong.
And I didnt get the use of always true while loop in the main method. Not required i guess.
Reply

Krishna says
JANUARY 20, 2018 AT 9:55 AM
Awesome article for this concept.
Reply

Tahere says
OCTOBER 23, 2017 AT 3:02 AM
thank you for your articles.
Reply

Priya says
JUNE 7, 2017 AT 4:57 AM
Will this program ever return two 50$ (or 20$ or 10$) notes. If yes , then can anybody explain how ?
Reply

ankit says
JANUARY 14, 2018 AT 2:15 AM
Yes it will..The first condition is for (50$)
if(cur.getAmount() >= 50){
int num = cur.getAmount()/50;
int remainder = cur.getAmount() % 50;
System.out.println("Dispensing "+num+" 50$ note");
consider cur.getAmount() is 156, so num will be 3 notes of $50
Reply

San Als says
JUNE 6, 2017 AT 2:31 AM
Thanks a lot mahn!!
Gave me a clear cut view about filter chain!
Keep up the good work!
Reply

Tommaso Resti says
NOVEMBER 28, 2016 AT 11:36 AM
Nice article.
Why don't you check if the next chain is set before to call his method dispense()?
Your example doesn't throw a null pointer exception just because you assert that the entered value is a multiple of 10. But this is an information that the 10DollarDispenser doesn't know.
Reply

Jigar Jarsania says
NOVEMBER 23, 2016 AT 3:12 AM

Great example...

You Rock always, Pankaj..!

Reply

Vaibhav M Nalwad says

NOVEMBER 22, 2016 AT 3:03 AM

You gave a crisp and Clear Explanation

Reply

Vaibhav M Nalwad says

NOVEMBER 22, 2016 AT 3:02 AM

I liked the Explanation.It was very crisp and clear

Reply

Karthik Garrepally says

JUNE 14, 2016 AT 5:05 AM

Good Example and quiet easy to understand the design pattern

Reply

Pankaj says

JUNE 14, 2016 AT 5:26 AM

Thanks Karthik for nice words.

Reply

Ricky Walia says

APRIL 23, 2016 AT 11:30 AM

nice and clear example. Thanks

Reply

Anusharmila says

JULY 7, 2015 AT 11:29 PM

Easy to understand. Thank you
Reply

Bharath says
JUNE 26, 2015 AT 12:12 AM
Thank you very much Pankaj. The tutorial on Design Patterns was very helpful. I could get a brief overview of Design Patterns and one more good point is that you have taken real world examples to demonstrate. In some part of the tutorial you have mentioned that one design pattern can be combined with the other to make it more efficient, it would be helpful if you post some samples on it. Great work buddy. Thanks a lot.
Reply

Muhammad says
JUNE 30, 2015 AT 1:58 PM
Yes this is very easy to follow and with real world examples. Thanks for sharing.
Reply

You Know says
DECEMBER 24, 2014 AT 5:51 AM
Mate, once again, check your English! It's loose coupling. Furthermore, Gaurav, you're right about only one object handling the request. That would be what's called a pure implementation. On the other hand, there's no real problem to let the request go through the entire tree.
Another point, which is not discussed in this article, is that you may set or change the next handler during runtime. You also may not know beforehand which are the handlers that should be used, allowing the client to set them as desired. As an example, you could have many Validator classes which can be called in a certain order. But that depends on the validation. So you apply CoR and make good use out of the possibility of combining different validators ordered in so many different ways.
Suppose you want to validate an email address. You need to check if the e-mail is correctly formed, if it exists, and if contains bad words. To do this job, you may have three different classes, and you can arrange them in any order – so you might as well let the client decide its own priority.
Reply

killer says
JULY 13, 2014 AT 8:55 AM
After implementing this design pattern for years, i commonly see that there is repeated code in the concrete classes just like the implementation of your dispense methods. The intent of this design pattern

is nice, but dont forget the DRY principle and it is a must in my humble opinion. I always use abstract classes for cor design pattern to eliminate duplicate code.
Reply

Gaurav says
JUNE 16, 2014 AT 12:30 PM
Good example. I'd like to point out one thing though. The pattern per GoF definition says that 'Only one object in the chain is supposed to handle a request'. In your example I see handlers($50 and $20) doing the work partially and letting $10 handler do the rest of the job.
Reply

Gaurav says
JUNE 16, 2014 AT 12:35 PM
I think I may be wrong coz GoF also says that – Use CoR when more than one object may handle the request…
Reply

kushi says
FEBRUARY 6, 2014 AT 8:54 AM
Hi.. Nice tutorial.. Just 1 suggestion setNextChain() would have been moved to DispenseChain abstract class instead of keeping DispenseChain as abstract class and overriding setNextChain() in all the concrete classes..
Reply

Pankaj says
FEBRUARY 6, 2014 AT 6:29 PM
By having setNextChain() as abstract method, we are forcing subclasses classes to provide implementation for that. For example purpose implementation is simple but there might be cases where some logic is involved for this.
Yes if it's simple as this, we can move it to abstract class. It's a design decision and should be based on project requirements.
Reply

Vijayendran T R says
JANUARY 6, 2014 AT 1:27 PM
Getting the below exception …
Exception in thread "main" java.lang.NullPointerException
at com.chain.responsibility.dispenser.HundredDollarDispenser.dispense(HundredDollarDispenser.java:26)
at com.chain.responsibility.dispenser.FiftyDollarDispenser.dispense(FiftyDollarDispenser.java:30)
at com.chain.responsibility.dispenser.ATMDispenseChain.main(ATMDispenseChain.java:33)
Reply

Anbarasan says
AUGUST 16, 2017 AT 11:04 AM
Problem is we don't have setNextPattern for the last level
Reply

jawahar says
OCTOBER 24, 2013 AT 12:11 AM
Hi,
Can you provide the download link for this example.
Reply

Pankaj says
OCTOBER 24, 2013 AT 2:38 PM
There is no download link, its simple java classes. You can copy paste the code and run it.
Reply

Graphic Design Singapore says
JULY 18, 2013 AT 4:30 PM
I savor, cause I discovered exactly what I was having a look for.
You've ended my 4 day long hunt! God Bless you man. Have a great day. Bye
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

GET IT ON Google Play

DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- Singleton
- Factory

- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial

Java Programming Courses

Chain Saw Sharpeners

Starting a Supply Chain

Top Supply Chain

Collaborative Supply Chain

Best Chain Saws

Replacement Chain

Learn Java in 7 Days