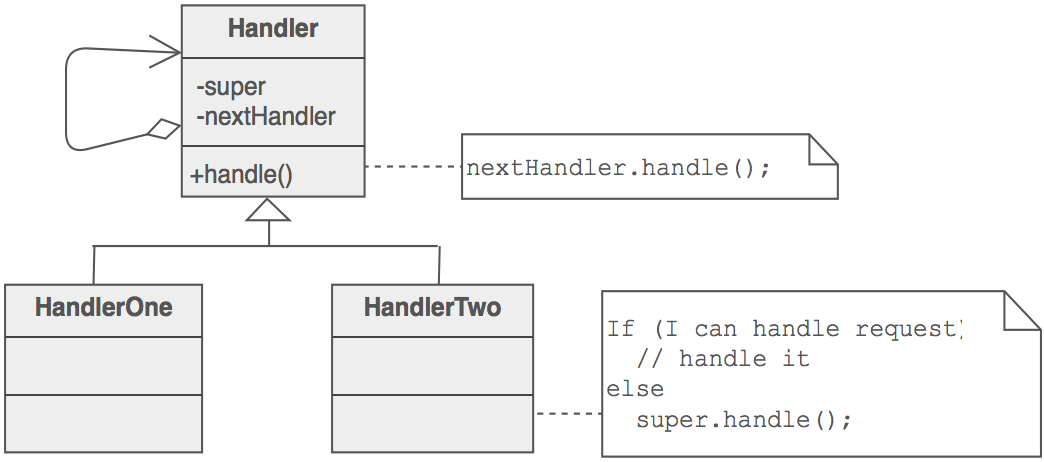
**Chain of Responsibility**

The chain of responsibility pattern is a behavioral object design pattern. In the chain of responsibility pattern, a series of handler objects are chained together to handle a request made by a client object. If the first handler can't handle the request, the request is forwarded to the next handler, and it is passed down the chain until the request reaches a handler that can handle the request or the chain ends. In this pattern, the client is decoupled from the actual handling of the request, since it does not know what class will actually handle the request.

In this pattern, a Handler is an interface for handling a request and accessing a handler's successor. A Handler is implemented by a Concrete Handler. The Concrete Handler will handle the request or pass it on to the next Concrete Handler. A Client makes the request to the start of the handler chain.

### Structure

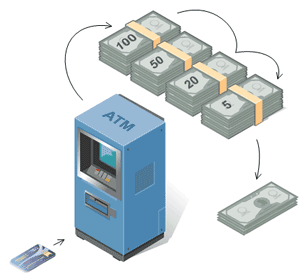
The derived classes know how to satisfy Client requests. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues.

. 

Multiple handlers could contribute to the handling of each request. The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a "null next".

### Example

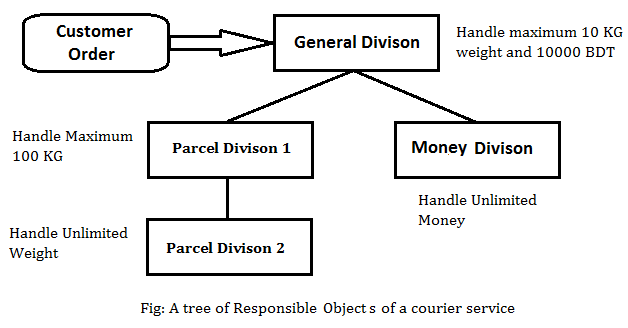
The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



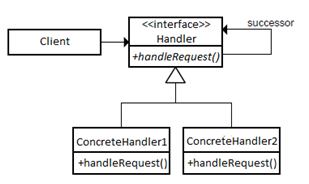
### Check list

1. The base class maintains a "next" pointer.
2. Each derived class implements its contribution for handling the request.
3. If the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer.
4. The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
5. The client "launches and leaves" each request with the root of the chain.
6. Recursive delegation produces the illusion of magic.

For an easy understanding let’s build a tree of responsible Divisions. In the root of the tree there should be**GeneralDivision**. This division can handle maximum weight of 10 Kilograms and 10000 dollars.**ParcelDivisionOne** can handle parcels of maximum weight 100 Kilograms. **ParcelDivisionTwo** can handle parcels of unlimited weight (imaginary) and the **MoneyDivision** can handle any amount of money. All of those are assumption. Whatever we assumed let’s take a look at the tree of Divisions:



**We will go through code segment but before that here is UML Class diagram of Chain of Responsibility Pattern. According to UML Class diagram it’s clear that there are three participants in this pattern and they are Request,RequestHandler and obviously there should be Client.**



## **Code**

**Let’s go through code segments. First of all we need to produce customer’s order (request object). The followingRequest class will produce request objects:**

class Request {

private double weight;

private double money;

public Request(double w, double m)

{

this.weight = w;

this.money = m;

}

public double getWeight()

{

return this.weight;

}

public double getMoney()

{

return this.money;

}

}

**Now I’m going to make an abstract class named Division. By extending this abstract class we will make responsible classes and each of those classes represents a particular Division of the courier service company. Let’s see what we have in Division abstract class.**

abstract class Divison {

protected List nextList = new ArrayList();

public void addNextDivison(Divison d)

{

this.nextList.add(d);

}

public void handle(Request request)

{

for(int i=0; i < nextList.size(); i++) {

Divison d = nextList.get(i);

d.handle(request);

}

}

protected abstract void processRequest();

}

**nextList is the list of next responsible objects. Doesn’t it make sense? Well, what will a Division (Responsible Object) do if it is unable to handle customer order (Request)? In such situation it will send orders to the divisions stored in nextList using the function handle(Request request)** **.**

**processRequest() is a abstract function which will be defined in concrete classes and this function will contain actual code to handle/process a request.**

**Now it’s time to define GeneralDivision**, **ParcelDivisionOne**, **ParcelDivisionTwo**, **MoneyDivision by extending Division.**

class GeneralDivison extends Divison {

@Override

public void handle(Request request)

{

if(request.getWeight() <= 10 && request.getMoney() <= 10000)

processRequest();

else super.handle(request);

}

public void processRequest()

{

System.out.println("This request is processed by -- General Divison");

}

}

class ParcelDivisonOne extends Divison {

@Override

public void handle(Request request)

{

if(request.getWeight() <= 100) processRequest();

else super.handle(request);

}

public void processRequest()

{

System.out.println("This request is processed by -- ParcelDivisonOne");

}

}

class ParcelDivisonTwo extends Divison {

@Override

public void handle(Request request)

{

if(request.getWeight() > 0) processRequest();

}

public void processRequest()

{

System.out.println("This request is processed by -- ParcelDivisonTwo");

}

}

class MoneyDivison extends Divison {

@Override

public void handle(Request request)

{

if(request.getMoney() > 0) processRequest();

}

public void processRequest()

{

System.out.println("This request is processed by -- MoneyDivison");

}

}

**Did we relate the responsible Divisions ever? Obviously we need to relate responsible objects whether it could be a chain or tree responsibility. Code in the following is easy to understand which does this relationship among responsible objects and also demonstrates how client will handle request using one responsible object.**

public class CourierServiceDemo {

private static Divison createDivison()

{

Divison div1 = new GeneralDivison();

Divison div2 = new ParcelDivisonOne();

Divison div3 = new ParcelDivisonTwo();

Divison div4 = new MoneyDivison();

div2.addNextDivison(div3);

div1.addNextDivison(div2);

div1.addNextDivison(div4);

return div1;

}

public static void main(String[] args)

{

Divison div = createDivison();

div.handle(new Request(101, 10));

}

}

**Now it’s your time to change request object data or make new request object and pass them to handle()function to see what happens.**

## **Applicability & Examples**

**Having so many design patterns to choose from when writing an application, it's hard to decide on which one to use, so here are a few situations when using the Chain of Responsibility is more effective:**

* **More than one object can handle a command**
* **The handler is not known in advance**
* **The handler should be determined automatically**
* **It’s wished that the request is addressed to a group of objects without explicitly specifying its receiver**
* **The group of objects that may handle the command must be specified in a dynamic way**

Here are some real situations in which the Chain of Responsibility is used:

### **Example 1**

**In designing the software for a system that approves the purchasing requests.**

**In this case, the values of purchase are divided into categories, each having its own approval authority. The approval authority for a given value could change at any time and the system should be flexible enough to handle the situation.**

**The Client in the example above is the system in need of the answer to the approval. It sends a request about it to an purchase approval authority. Depending on the value of the purchase, this authority may approve the request or forward it to the next authority in the chain.**

**For example let’s say a request is placed for the purchase of a new keyboard for an office. The value of the purchase is not that big, so the request is sent from the head of the office to the head of the department and then to the materials department where it stops, being handled locally. But if equipment for the whole department is needed then the request goes form the head of the department, to materials department, to the purchase office and even to the manager if the value is too big.**

### **Example 2**

**In designing the software that uses a set of GUI classes where it is needed to propagate GUI events from one object to another.**

**When an event, such as the pressing of a key or the click of the mouse, the event is needed to be sent to the object that has generated it and also to the object or objects that will handle it.**

**The Client is, of course, the object that has generated the event, the request is the event and the handlers are the objects that can handle it. So, if we have a handler for the click of the mouse, a handler for the pressing of the ‘Enter’ key and a handler for the pressing of the ‘Delete’ key, that is the chain of handlers that take care of the events that are generated.**

### **Example 3**

**In designing a shipping system for electronic orders.**

**The steps to complete and handle the order differs form one order to another based on the customer, the size of the order, the way of shipment, destination and more other reasons. The business logic changes also as special cases appear, needing the system to be able to handle all cases.**

**The Client, the electronic order in process, requests shipping based on a set of pieces of information. Its request is turned by the system into a specific form, combining the steps to completing and the details of handling, based on the input information. The system will send this type of request through a chain of order-handlers until the input information that it comes with matches the input the order-handles takes. When special cases appear, all that is needed is a new handler to be added in the chain.**

## **Specific problems and implementation**

**The classic implementation of the Chain of Responsibility is just the first step in applying the pattern to our own application. Improvements based on the type of commands we are handling are needed, in order to make the use of this pattern effective.**

### **Representing requests**

**In real life each handler represents a system. And each system can handle specific requests or requests common to more handlers. We should take this issue in consideration when we implement this pattern. In the classical samples of the CoR found on the net you can see that the request is generally represented by an integer. Of course in real life we can not use primary data types as a request.**

**A clever design should be a flexible one. The best solution here is to create an interface a super class Request (or and interface) where to the default behavior. Then if we need to add a new handler and a specific request all we need is to extend the Request base class.**

**Of course this is not the only approach. Let’s consider the shipping system example. Each request will have to contain a large amount of data. Creating request examples for this might be difficult. We can take some xml objects containing the data, generated during the application flow (let’s assume we already have the code implemented for that) and pass them to each handler.**

**Or since the data was already saved in the database (let’s assume that also) we can pass only the id’s of the involved objects and then each handler will take the data required from db.**

### **Use on existing code**

**The last, but not least problem that the Chain of Responsibility creates to a programmer is the fact that it is impossible to introduce the pattern into the existing classes without modifying the source code and, even in the case where the pattern is already included in the code, if new operations need to be added to the Handler, it is impossible to do that without modifying the source code. So the basic idea is to decide from the start on whether to use the pattern or not and if we do, what methods we need.**

## **Hot points**

* **The fundamental flaw of the pattern is the fact that it gets easily broken: if the programmer forgets to call the next handler in the concreteHandler the request gets lost on the way. This problem comes from the fact that the execution is not handled entirely by the superclass and the call is triggered in the superclass.**
* **When implementing the CoR pattern a special care should be taken for the request representation. The request is not considered a distinctive part of the CoR pattern, but it is still used in all the components of the pattern.**
* **Another flaw of the Chain of Responsibility is the fact that some requests may end up unhandled due to the wrong implementation of concrete handler, their propagation slowing down the rest of the application. This means that extra care is needed when taking into account the requests that may appear in the process.**

### **Reference implementations in JDK**

* [**javax.servlet.Filter#doFilter()**](http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse,%20javax.servlet.FilterChain%29)
* **The doFilter method of the Filter is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. The FilterChain passed in to this method allows the Filter to pass on the request and response to the next entity in the chain.**
* [**java.util.logging.Logger#log**](http://docs.oracle.com/javase/6/docs/api/java/util/logging/Logger.html#log%28java.util.logging.Level,%20java.lang.String%29)
* **If the logger is currently enabled for the given message level then the given message is forwarded to all the registered output Handler objects.**