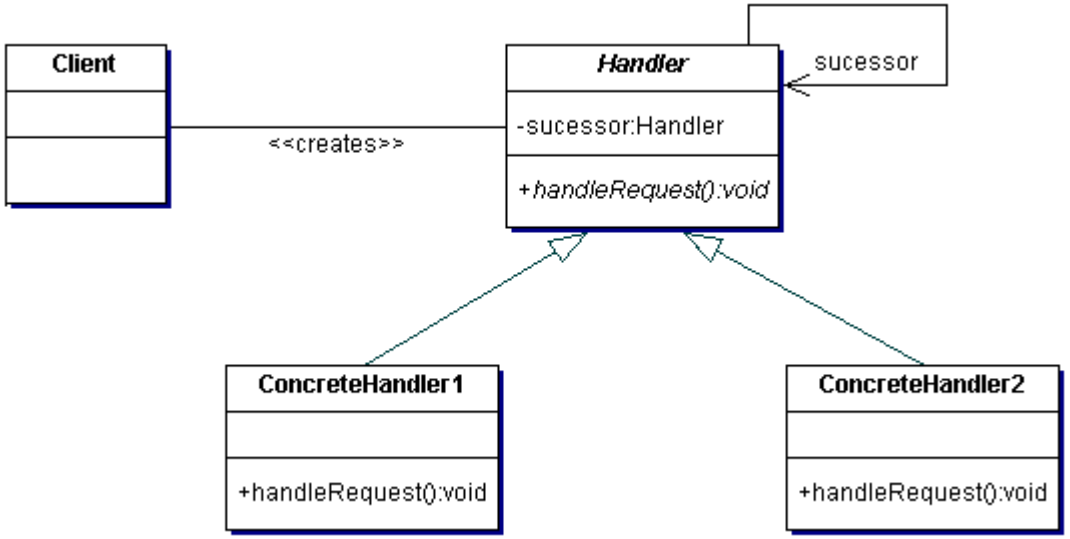
Chain Of Responsibility

- A behavioural pattern consisting of a source of command objects and a series of processing objects
- Utilizes the Single Responsibility Principle of OO
- Can have a linked list feel to it, where one object in the chain has a reference to another object in the chain.
- The chain can even be represented in tree form, when commands can take a variety of 'directions', thus a 'tree of responsibility'

Chain of Responsibility UML

UML again :-)


Client
Handler
sucessor
-sucessor:Handler
+handleRequest():void
<<creates>>
ConcreteHandler1
+handleRequest():void
ConcreteHandler2
+handleRequest():void

Participants

- Handler - defines an interface for handling requests
- RequestHandler - handles the requests it is responsible for
  - If it can handle the request it does so, otherwise it sends the request to its successor
- Client - sends commands to the first object in the chain that may handle the command

How It Works

- The Client in need of a request to be handled sends it to the chain of handlers, which are classes that extend the Handler class.
- Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client.
- If ConcreteHandler_i can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler_i+1, the next one in the chain.
- Typically, only one handler is needed to handle a given request (thus others are ignored once the request is handled)

Code Example: Handler Interface

```
abstract class PurchasePower {

    protected final double base = 500;
    protected PurchasePower successor;

    public void setSuccessor(PurchasePower successor) {
        this.successor = successor;
    }

    abstract public void processRequest(PurchaseRequest
request);

}
```

Concrete Handler 1

```
class ManagerPPower extends PurchasePower {

    private final double ALLOWABLE = 10 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("Manager will approve $"+
request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
        }
    }

}
```

Concrete Handler 2

```
class DirectorPPower extends PurchasePower {

    private final double ALLOWABLE = 20 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("Director will approve $"+
request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
        }
    }

}
```

Concrete Handler 3

```
class VicePresidentPPower extends PurchasePower {

    private final double ALLOWABLE = 40 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("Vice President will approve $" +
request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
        }
    }
}
```

Concrete Handler 4

```
class PresidentPPower extends PurchasePower {

    private final double ALLOWABLE = 60 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("President will approve $" +
request.getAmount());
        }
        else {
            System.out.println( "Your request for $" +
request.getAmount() + " needs a board meeting!");
        }
    }

}
```

Class to Hold Request Data (for example)

```
class PurchaseRequest {

    private int number;
    private double amount;
    private String purpose;

    public PurchaseRequest(int number, double amount, String purpose) {
        this.number = number;
        this.amount = amount;
        this.purpose = purpose;    }

    public double getAmount() {
        return amount;    }
    public void setAmount(double amt)  {
        amount = amt;    }

    public String getPurpose() {
        return purpose;    }
    public void setPurpose(String reason) {
        purpose = reason;    }

    public int getNumber(){
        return number;    }
    public void setNumber(int num) {
        number = num;    }   }
```

Client

```
class CheckAuthority {

    public static void main(String[] args) {
        ManagerPPower manager = new ManagerPPower();
        DirectorPPower director = new DirectorPPower();
        VicePresidentPPower vp = new VicePresidentPPower();
        PresidentPPower president = new PresidentPPower();
        manager.setSuccessor(director);
        director.setSuccessor(vp);
        vp.setSuccessor(president);

        //enter ctrl+c to kill.
        try{
            while (true) {
                System.out.println("Enter the amount to check who should approve your
expenditure.");
                System.out.print(">");
                double d = Double.parseDouble(new BufferedReader(new
InputStreamReader(System.in)).readLine());
                manager.processRequest(new PurchaseRequest(0, d, "General"));   }       }
        catch(Exception e){
            System.exit(1);       }      } }
```

When to Use

- 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

Example Usage Scenario

In designing software that uses a set of GUI classes where it is necessary 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 needs 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.

Another Example Scenario

In designing a shipping system for electronic orders.

The steps to complete and handle the order differs from one order to another based on the customer, the size of the order, the way of shipment, destination, etc. The business logic also changes as special cases appear, requiring 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 of 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-handler takes. When special cases appear, all that is needed is a new handler to be added in the chain.

Gotchas with Chain of Responsibility

- The pattern is easily broken if programmer forgets to call the next handler in the chain – happens most often when adding a new handler to the chain
- If request object is not designed properly, handlers in the chain may have more work to do or difficulty doing what they need to do
- Requests may go unhandled if implementation in concrete handler is incorrect (logic errors)
- Careful thought must be placed into how to decide to handle each request within each handler