Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

abdulrahman@nu.edu.pk

alphapeeler.sf.net/pubkeys/pkey.htm

pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

abdulmahmood-sss alphasecure

armahmood786

http://alphapeeler.sf.net/me

alphapeeler#9321

reddit.com/user/alphapeeler

www.flickr.com/alphapeeler

http://alphapeeler.tumblr.com

armahmood786@jabber.org

alphapeeler@aim.com

mahmood_cubix 48660186

alphapeeler@icloud.com

pinterest.com/alphapeeler

www.youtube.com/user/AlphaPeeler

Chain of Responsibility Design Pattern

What is Proxy pattern?

Proxy is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Chain of Responsibility

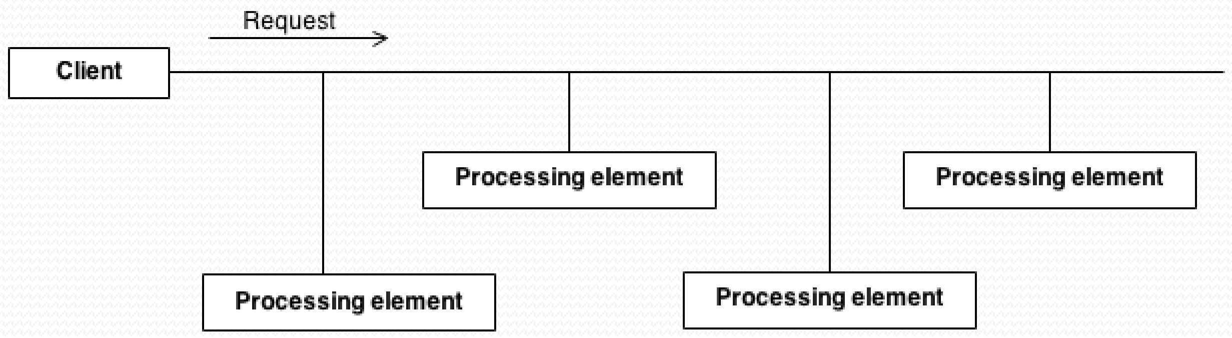
- **Intent**
- CR pattern is used to achieve loose coupling in where a request from the client is passed to a chain of objects to process them.
- Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
- The handler is determined at runtime.
- Please note that a request not handled at all by any handler is a valid use case.

Chain of Responsibility

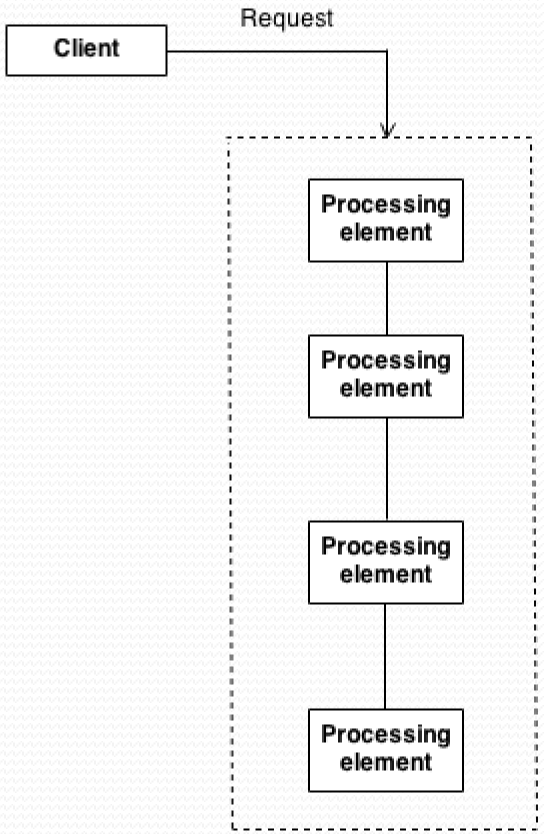
- **Where and When CR pattern is applicable :**
- When you want to <u>decouple</u> a request's sender and receiver
- Multiple objects, determined at <u>runtime</u>, are candidates to handle a request
- When you don't want to specify handlers <u>explicitly</u> in your code
- When you want to issue a request to one of several objects without specifying the receiver <u>explicitly</u>.

Chain of Responsibility

- **Problem**
- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.
-



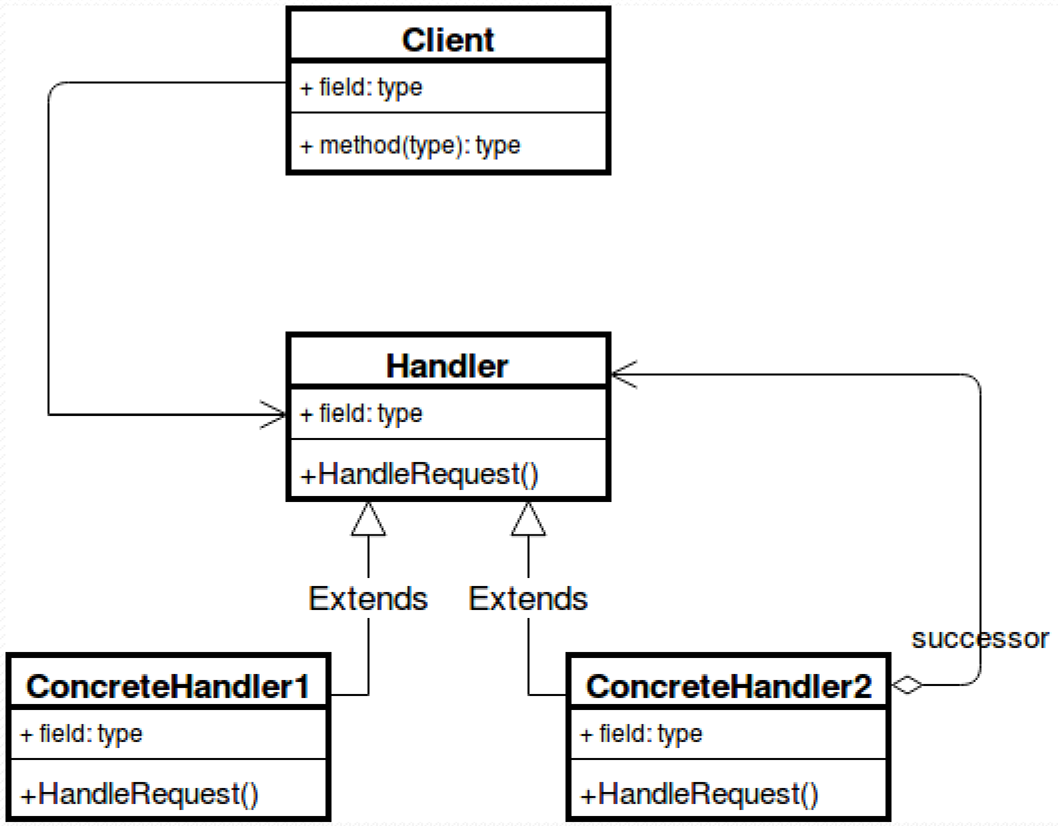
Chain of Responsibility

- **Discussion**
- Encapsulate the processing elements inside a "pipeline" abstraction; and have clients "launch and leave" their requests at the entrance to the pipeline.
- 

Chain of Responsibility

- **Structure**

•**Handler :** This can be an interface which will primarily recieve the request and dispatches the request to chain of handlers. It has reference of only first handler in the chain and does not know anything about rest of the handlers.
•**Concrete handlers :** These are actual handlers of the request chained in some sequential order.
•**Client :** Originator of request and this will access the handler to handle it.



Ex. 1 : Chain of Responsibility

```java
public interface Chain {
    public abstract void setNext(Chain nextInChain);
    public abstract void process(Number request);
}
public class Number {
private int number;
    public Number(int number) {
        this.number = number;
    }
    public int getNumber() {
        return number;
    }
}
public class NegativeProcessor implements Chain {
private Chain nextInChain;
    public void setNext(Chain c) {
        nextInChain = c;
    }
    public void process(Number request) {
        if (request.getNumber() < 0) {
            System.out.println("NegativeProcessor : " + request.getNumber());
        } else {
            nextInChain.process(request);
        }
    }
}
```

Ex. 1 : Chain of Responsibility

```
public class ZeroProcessor implements Chain {
private Chain nextInChain;
    public void setNext(Chain c) {
        nextInChain = c;
    }
    public void process(Number request) {
        if (request.getNumber() == 0) {
            System.out.println("ZeroProcessor : " + request.getNumber());
        } else {
            nextInChain.process(request);
        }
    }
}
  public class PositiveProcessor implements Chain {
  private Chain nextInChain;
      public void setNext(Chain c) {
          nextInChain = c;
      }
      public void process(Number request) {
          if (request.getNumber() > 0) {
              System.out.println("PositiveProcessor : " + request.getNumber());
          } else {
              nextInChain.process(request);
          }
      }
  }
```

Ex. 1 : Chain of Responsibility

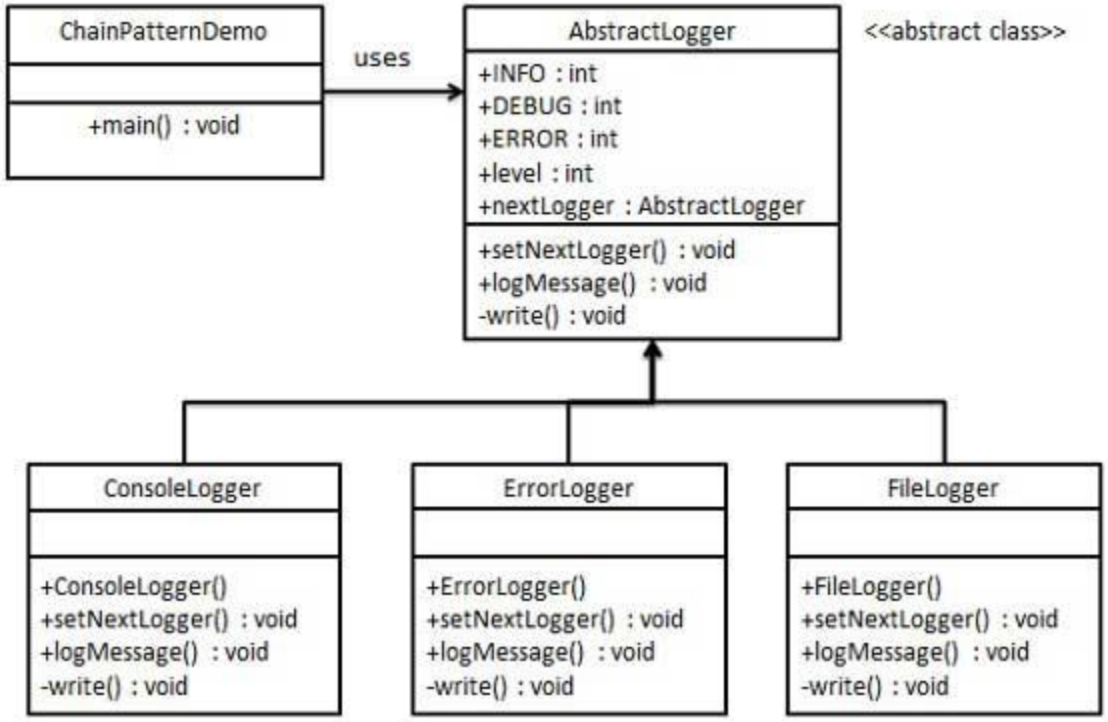
```
public class TestChain {
public static void main(String[] args) {
//configure Chain of Responsibility
        Chain c1 = new NegativeProcessor();
        Chain c2 = new ZeroProcessor();
        Chain c3 = new PositiveProcessor();
        c1.setNext(c2);
        c2.setNext(c3);

        //calling chain of responsibility
        c1.process(new Number(90));
        c1.process(new Number(-50));
        c1.process(new Number(0));
        c1.process(new Number(91));
}
}
```

- Output:

```
PositiveProcessor : 90
NegativeProcessor : -50
ZeroProcessor : 0
PositiveProcessor : 91
```

Ex. 2 : Chain of Responsibility

- **Implementation**
- We have created an abstract class *AbstractLogger* with a level of logging. Then we have created three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Ex. 2 : Chain of Responsibility

```
public abstract class AbstractLogger {
 public static int INFO = 1;
 public static int DEBUG = 2;
 public static int ERROR = 3;
 protected int level;
 //next element in chain or responsibility
 protected AbstractLogger nextLogger;
 public void setNextLogger(AbstractLogger nextLogger){
    this.nextLogger = nextLogger;
 }
 public void logMessage(int level, String message){
    if(this.level <= level){
       write(message);
    }
    if(nextLogger !=null){
       nextLogger.logMessage(level, message);
    }
 }
 abstract protected void write(String message);
}
```

Ex. 2 : Chain of Responsibility

```
public class ConsoleLogger extends AbstractLogger {
public ConsoleLogger(int level){
        this.level = level;
    }
    @Override
    protected void write(String message) {
        System.out.println("Standard Console::Logger: " + message);
    }
}

public class FileLogger extends AbstractLogger{
public FileLogger(int level){
        this.level = level;
    }
    @Override
    protected void write(String message) {
        System.out.println("File::Logger: " + message);
    }
}

public class ErrorLogger extends AbstractLogger{
public ErrorLogger(int level){
        this.level = level;
    }
    @Override
    protected void write(String message) {
        System.out.println("Error Console::Logger: " + message);
    }
}
```

Ex. 2 : Chain of Responsibility

```
public class ChainPatternDemo {
private static AbstractLogger getChainOfLoggers(){
    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
    AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
    AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
    errorLogger.setNextLogger(fileLogger);
    fileLogger.setNextLogger(consoleLogger);
   return errorLogger;
}
public static void main(String[] args) {
AbstractLogger loggerChain = getChainOfLoggers();
    loggerChain.logMessage(AbstractLogger.INFO, "Info");
    System.out.println();
    loggerChain.logMessage(AbstractLogger.DEBUG, "Debug info");
    System.out.println();
    loggerChain.logMessage(AbstractLogger.ERROR, "Error info");
}
}
```

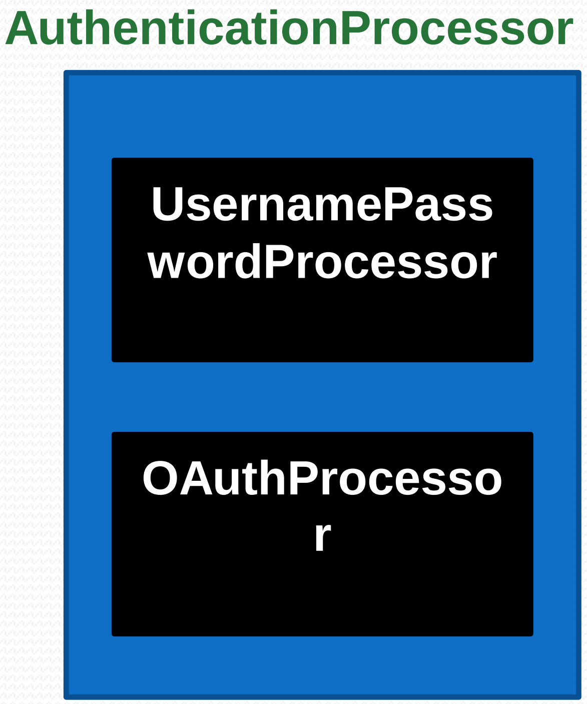
```
Output:
Standard Console::Logger: Info

File::Logger: Debug info
Standard Console::Logger: Debug info

Error Console::Logger: Error info
File::Logger: Error info
Standard Console::Logger: Error info
```

Pipeline: Error -> File -> Console

Ex3

- We're going to use Chain of Responsibility to create a chain for handling authentication requests.

- https://www.baeldung.com/chain-of-responsibility-pattern


AuthenticationProcessor
UsernamePasswordProcessor
OAuthProcessor