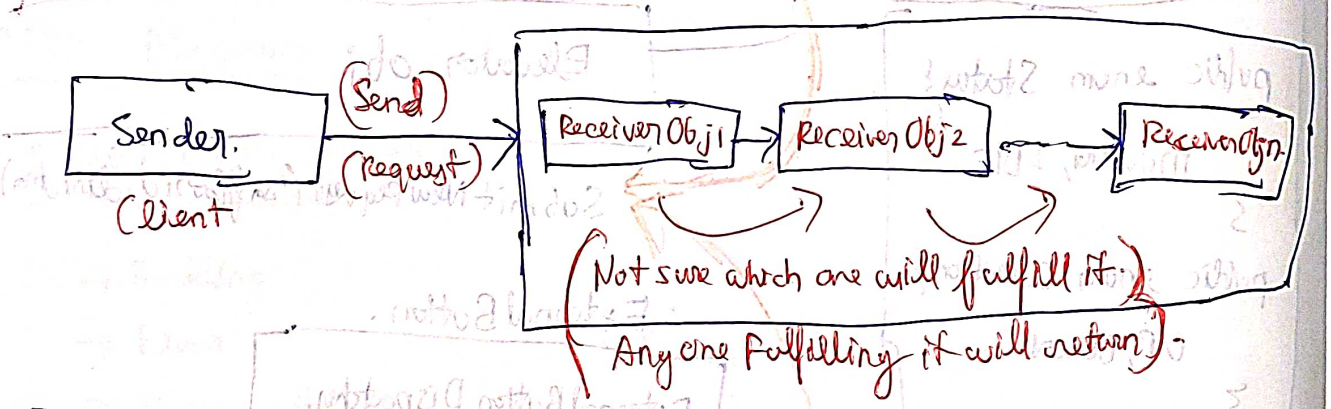
Chain of Responsibility Design Pattern.

Application Usage:-

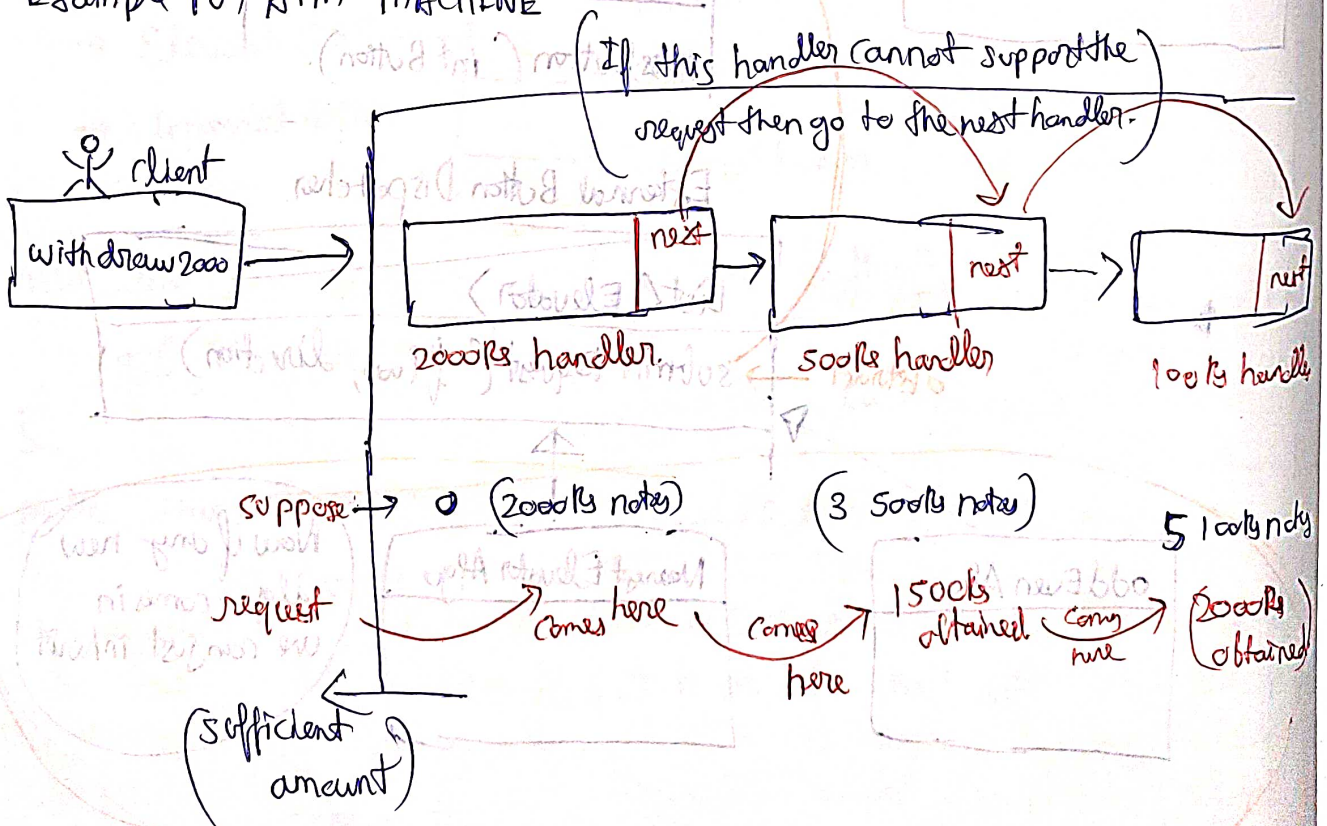
→ ATM / Vending Machine

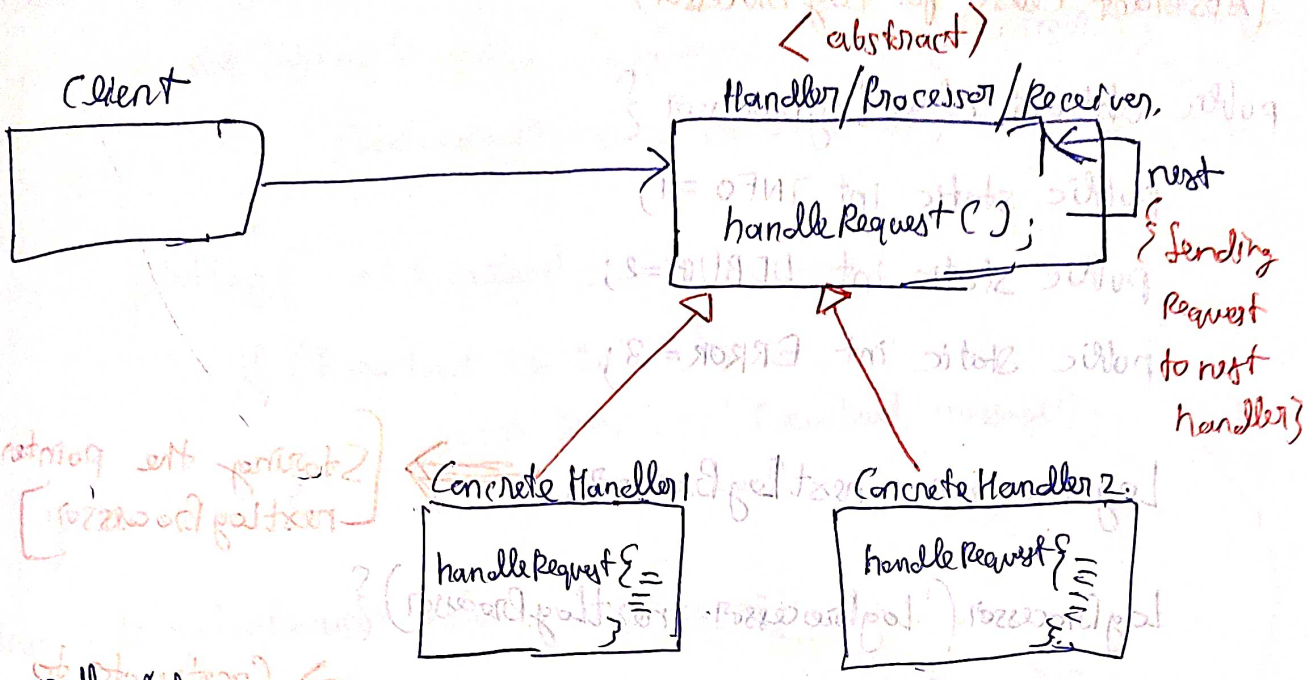
→ Design Logger (Amazon)

This design pattern is used when client sends a request, and it does not care which receiver will get the request and will give the response.



Example for ATM MACHINE





Generally a logger will look like,

```
Logger obj = new Logger();
obj.log( Info , "msg");
obj.log( Debug, "msg");
obj.log( Error, "msg");
```

If you notice here were are note telling which implementation of logger are we calling. Is it Info, Is it ~~msg~~ Debug, Is it Error.

```
Log Processor {
public static int INFO = 1;
public static int DEBUG = 2;
public static int ERROR = 3;

LogProcessor nextLogProcessor;

LogProcessor (LogProcessor nextLogProcessor) {
    this.nextLogProcessor = nextLogProcessor;
}
```

The plan ~~hai~~ is to keep ~~the~~ next logger object and make the chain of Debuggers. such that call happens in a chain format and reaches destination.

(Abstract class for LogProcessor)

```
public abstract class LogProcessor {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    LogProcessor nextLogProcessor;   ⟹ [Storing the pointer to nextLogProcessor.]

    LogProcessor(LogProcessor nextLogProcessor){
        this.nextLogProcessor = nextLogProcessor;   ⟹ Constructor to initialize next LogProcessor.
    }

    public void log(int logLevel, String message){
        ~~if (nextLogger != Null){~~
        ~~next~~
        if (nextLogProcessor != null){
            nextLogProcessor.log(logLevel, message);
        }
    }
}
```

(Child Classes)

```
class ErrorLogProcessor extends LogProcessor {

    ErrorLogProcessor (LogProcessor nextLoggProcessor){
        super.~~log~~ nextLogProcessor = nextLogProcessor;
    }
    printLog(int ErrorLevel, String message){
        if (ErrorLevel != Error) { super.printLog(ErrorLevel, message);
        }
    }
}
```

```
class InfoLogProcessor extends LogProcessor {
    public InfoLogProcessor ( LogProcessor nextLogProcessor ) {
        super.nextLogProcessor = nextLogProcessor;
    }
    printLog ( int ErrorLevel, string message ) {
        if (ErrorLevel != Info) {
            super.printLog ( ErrorLevel, message);
        }
    }
}

class DebugLogProcessor extends LogProcessor {
    DebugLogProcessor ( LogProcessor nextLogProcessor) {
        super.nextLogProcessor = nextLogProcessor.
    }
    printLog ( int ErrorLevel, string message) {
        if ( ErrorLevel != DEBUG) {
            super.printLog (ErrorLevel, message);
        }
    }
}
```

(Info → DEBUG → ERROR)

```
main( ) {
    LogProcessor logProcessor = new InfoLogProcessor( new DebugLogProcessor( new ErrorLogProcessor (null)));
    InfoLogProcessor( Debug, "Print Debug")
    InfoLogProcessor. printLog ( Debug, "Print Debug");
```

Look Next Page for flow

public void main(){

Where you are defining super for each other

~~Logger~~

LogProcessor Logger = new ~~Log~~InfoLogProcessor(
new DebugLogProcessor(
new ErrorLogProcessor(null)));

Logger.printLog(Error, "this is an Error").

This in INFOLOGGER

So (ERROR != INFO)

So we call super.printLog(ERROR, "this is an ERROR")

super = DebugLogProcessor.

DebugLogProcessor.printLog(ERROR, "this is ERROR")

this is DEBUGLOGGER

So

(ERROR != DEBUG)

So we call super.printLog(ERROR, "this is an ERROR").

super = ERRORLOGGER

ERRORLOGGER.printLog(ERROR, "this is ERROR")

So (ERROR == ERROR)

Print("this is an ERROR").