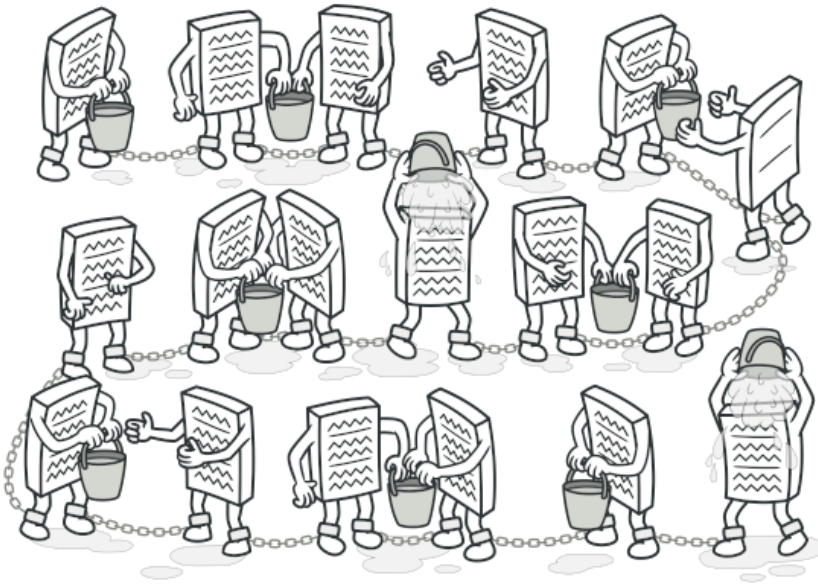
Chain of Responsibility

A Behavioral Pattern

Gruppe 19

23-04-2019



Figur 1 - https://refactoring.guru/images/patterns/content/chain-of-responsibility/chain-of-responsibility.png

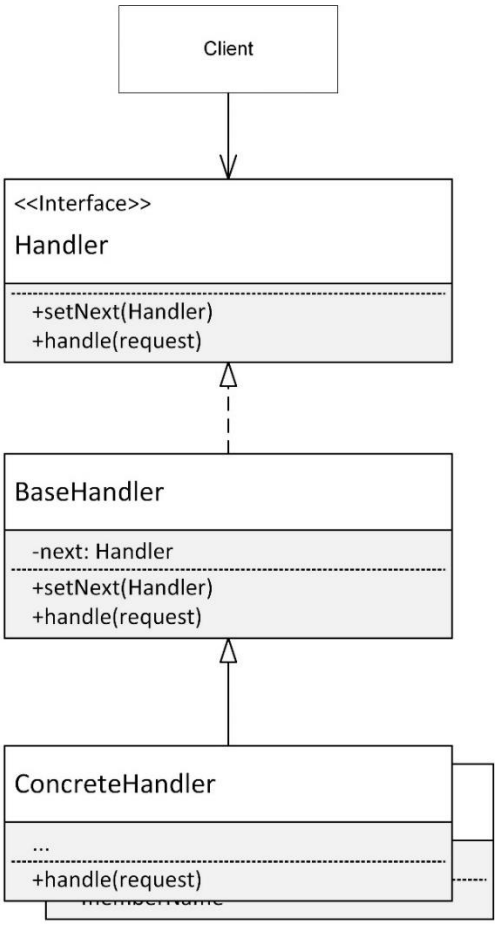
auID	Name
au580521	Mathias Thomassen
au580513	Thomas Møller
au586626	Valdemar Tang
au575348	Oskar Thorin Hansen

AARHUS UNIVERSITET
INGENIØRHØJSKOLEN

1 Indhold

2 Chain of Responsibility Pattern

Chain of Responsibility (COR) er et behavioral design pattern, hvilket vil sige, at det bliver brugt til interaktion og ansvar i forhold objekter. COR gør det muligt at videregive requests i en kæde af handlers. Man kan sige at det er en linked list af handlers. Når en af disse handlere får requesten, så kan den vælge at gøre med noget med, og/eller sende den videre til den næste handler i kæden. Dette giver at man undgår kobling mellem klienten, som sender en request og receiverne, som i dette tilfælde er handlere.



Figur 2 - UML for COR strukturen

På Figur 2 ser man et UML-diagram over en basale struktur af COR. Handler definerer det fælles interface til alle handlers, og indeholder de metoder, som skal implementeres i handlers, som client, kan gøre brug af.

BaseHandler-klassen fungerer, som base klasse til concretehandler, og indeholder interface implementeringen. Denne klasse bliver brugt til at lagre referencerne til den næste handler i COR. Client kan således bruge setNext, for at tilføje en handler til COR. Derudover kan klassen også bruge til at tjekke om COR er tom, og sørge for at den næste i kæden bliver kaldt.

ConcreteHandler indeholder så den egentlige kode, som bliver brugt til at håndtere requesten, som passerer ned gennem kæden. Handleren skal så vælge om den vil gøre noget ved requesten og/eller om den blot vil sende den videre i kæden, og kalder herved base.handle på requesten.

Når man så lader handlerne vælge om den vil sende requesten videre, så kan der være nogle konsekvenser. Hertil kan man ende op med, at nogle requests ikke kommer igennem til nogle handlers, som egentlig skulle bruge requesten. Derudover kan man også ende med at en request kommer igennem listen uden at være handlet. Til gengæld kan det være en fordel, at man selv vælger rækkefølgen handlerne har mulighed for at håndtere request.

Man bruger som regel kun COR, når det er relevant i forhold til kæder af objekter, som alle vil benytte sig af en request, samt at man ønsker dem i en bestemt rækkefølge. Dette medfører også at COR ikke er så ofte brugt, da det kun er i meget bestemte tilfælde, hvor det viser sig at være nyttigt.

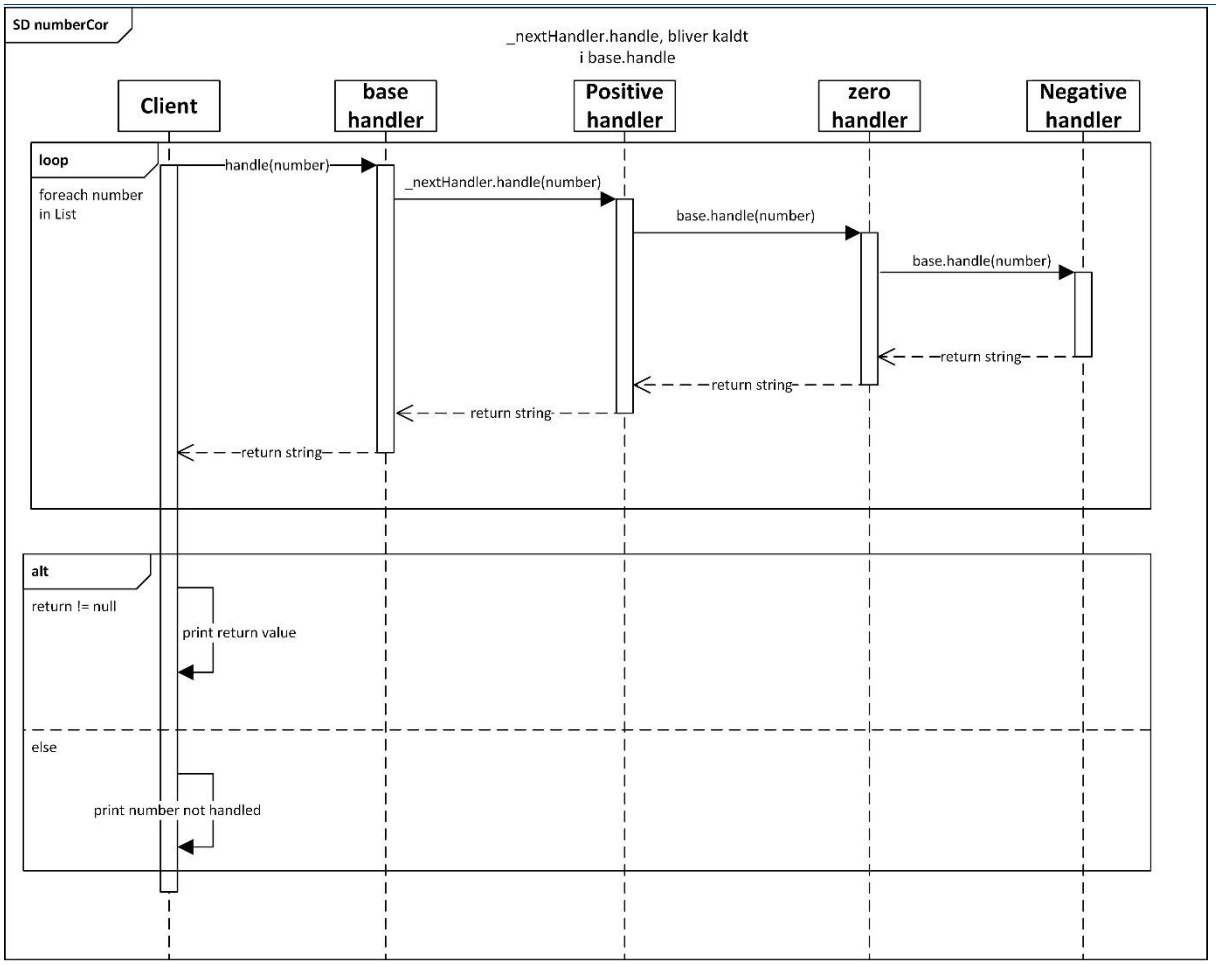
3 Eksempler

For at beskrive COR bedre gennemgår vi et eksempel. Dette eksempel er meget simpel, hvor vi laver en request på et tal, og så oprettes 3 handlers, som skal vælge om de vil handle requesten eller sende den videre.

Derudover har vi lavet et andet eksempel, som tager udgangspunkt i ATM system fra SWT-undervisningen. Systemet er blevet designet til at bruge chain of responsibility, da man modtager noget data fra en transponder.dll, som man skal handle i en bestemt rækkefølge. Dette er en oplagt mulighed for at benytte Cor.

3.1 Eksempel 1 – Positive, Zero or Negative request.

I dette eksempel anvendes COR til at bestemme om et nummer er positivt, negativt eller 0. Til dette anvendes 3 handlers som skal håndtere requestet eller sende det videre. Her stoppes kæden hvis én Handler ”fanger” requestet. På sekvensdiagrammet på Figur 3. kan man se, at client kalder basehandle med en request. Base klassen kalder så nexthandle i Cor, som i dette tilfælde er den første handler i Cor. Dette er så positive handler, hvis tallet så er positivt, så returner positive handler, og kalder ikke base.handle med requesten. Hvis tallet ikke er et positivt tal, så kaldes derimod base.handle, uden at der bliver returneret til client. Dette foregår så recursivt indtil en af handlerne returner en string, som så siger at tallet er blevet håndteret af denne handler.

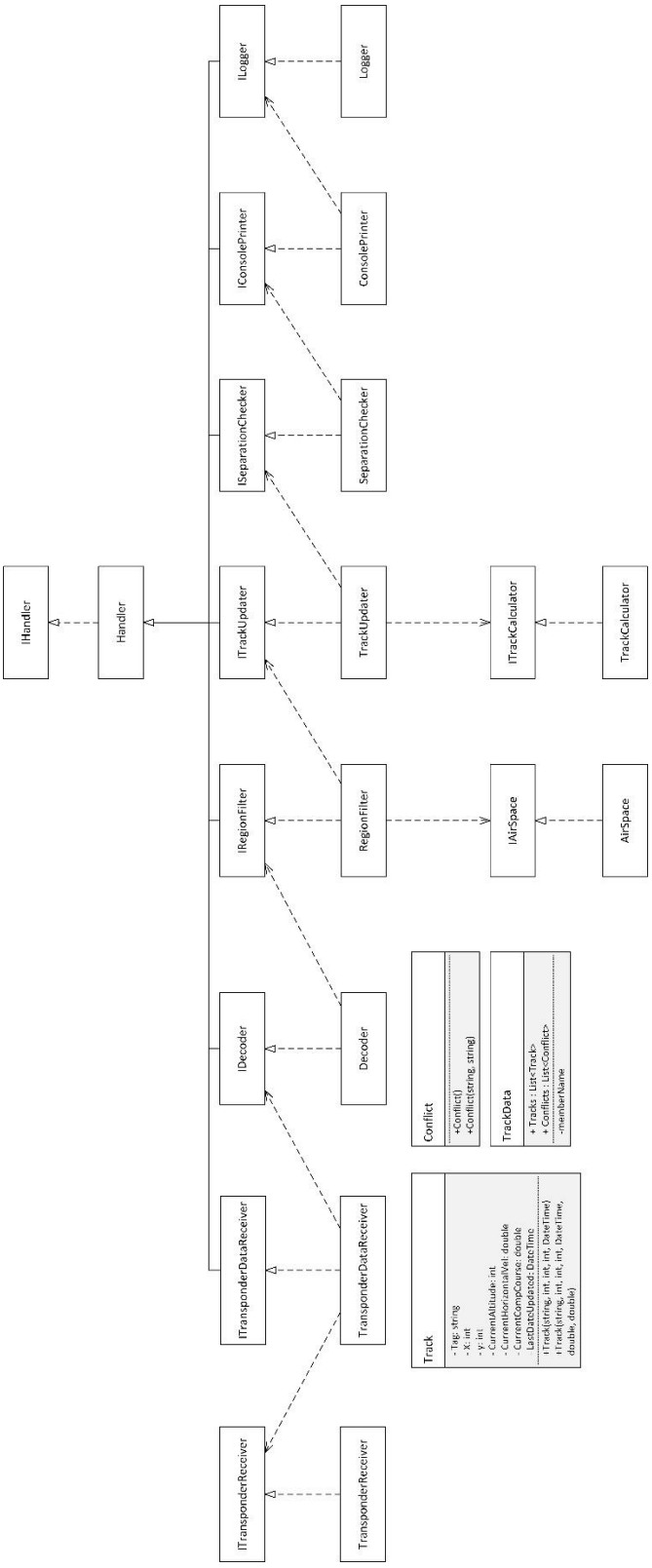


Figur 3 - Sekvensdiagram for numberCor eksempel

En gennemgang af eksemplet kan ses i præsentationsvideoen.

3.2 Eksempel 2 – ATM system

For at demonstrere, hvordan COR kunne bruges i et lidt mere avanceret system, er Air Traffic Monitor opgaven fra SWT blevet modificeret, så denne benytter sig af COR. I denne opgave skulle der laves et system, som holder øje med sit eget Air Space, og tracker fly, som er inde i dette Air Space. Disse tracks udskrives så til konsollen, med navn, koordinater, højde og hastighed. Hvis 2 tracks kommer for tæt på hinanden, skulle dette skrives ud til en logfil, samt skrives ud på konsollen. Efter vi fik feedback på opgaven, indså vi at det kunne løses på en smartere måde. Dataene skal processeres af en masse forskellige led, hvortil COR er oplagt at bruge. Designet blev da gentænkt således at vi kunne inkorporere COR. Designet kan ses på Figur 4. Det er her værd at notere at dataene godt kan håndteres/modificeres af flere handlers, i modsætning til det tidligere eksempel hvor dataene kun blev håndteret af én Handler.



Figur 4 - Design af Air Traffic Monitor med Chain of Responsibility pattern. De fleste medlemsfunktioner er ikke medtaget for overskuelighedens skyld. Track, Conflict og TrackData er klasser som blot skal indeholde data. Associationerne hertil er ikke vist for overskuelighedens skyld.

Fly dataene kommer fra TransponderReceiver'en, som blev udleveret sammen med opgaven. Disse datasæt skal igennem en masse forskellige klasser, hvor det bliver behandlet. Det er altså her oplagt at bruge COR, da vi har en client som sender noget data, som skal igennem en lang kæde, hvor hvert led har sin egen opgave med datasættet. Hvis vi undervejs finder ud af at datasættet ikke skal videre i kæden, gør COR det muligt at kunne bryde kæden, så vi ikke unødvendigt skal igennem hele kæden. Datasættet modtages som en streng, og det første der skal gøres, er at konverterer dette om til en Track, så dataene er nemmere at arbejde på. Dette sørger Decoder klassen for. Herefter bliver datasættet sendt videre til RegionFilter klassen, som undersøger om Tracket er indenfor systemets AirSpace. Hvis Tracket ikke er inden for AirSpace, er der ingen grund til at sende datasættet videre, og vi bryder her kæden. Hvis Tracket er inden for AirSpace, sendes datasættet videre i kæden til TrackUpdater, som sørger for at opdatere listen med Tracks. Hvis Tracket ikke findes i listen, tilføjes den til listen, men hvis den findes i forvejen, opdateres det eksisterende Track. Datasættet sendes videre til SeparationChecker, som undersøger om 2 Tracks i listen er for tættet på hinanden. Hvis der er det, laves der en liste med disse Tracks, som sendes med videre i kæden, sammen med listen af Tracks. Hvis der ikke er 2 Tracks der er tæt på hinanden, er det kun listen af Tracks der sendes videre. Nu modtager ConsolePrinteren datasættet, som sørger for at udskrive dette til konsollen, og sender det så videre til Logger. Denne sørger for at skrive til en logfil, hvis der er 2 Tracks der er tæt på hinanden, og ellers gør den ingen ting. Dermed har datasættet været hele kæden igennem. Det er her også vigtigt at de forskellige handlers sættes i en korrekt logisk rækkefølge, så vi minimerer mængden af data der sendes samt at hver handler får noget korrekt data at arbejde på.

Hvert led i kæden skal implementere Handler-klassen og override Handle() funktionen. Implementeringen af Handler og IHandler kan ses på Figur 5.

```
public interface IHandler
{
    IHandler SetSuccessor(IHandler handler);
    void Handle(object data);
}
```

```
public abstract class Handler : IHandler
{
    protected IHandler _successor;

    public IHandler SetSuccessor(IHandler successor)
    {
        _successor = successor;
        return _successor;
    }

    public virtual void Handle(object data)
    {
        _successor?.Handle(data);
    }
}
```

Figur 5 - Handler og IHandler implementerin

Efter disse to klasser er blevet oprettet, skal hvert led i kæden nedarve og implementere Handle() funktionen fra Handler klassen. Et eksempel på dette kan ses på Figur 5.

```
public override void Handle(object data)
{
    _tracks = new List<Track>();
    foreach (var track in (List<string>)data)
    {
        //Decode
        List(track);
        TypeConverter();

        Track newTrack = new Track(flightNum, x, y, altitude:alt, dateTimeNew);
        _tracks.Add(newTrack);
    }

    if (_tracks.Count == 0) //Break chain if no data
    {
        Console.Clear();
        Console.WriteLine(value:"No airplanes currently in airspace");
    }
    else
    {
        base.Handle(new List<Track>(_tracks));
    }
}
```

```
public override void Handle(object data)
{
    _tracks = new List<Track>();
    foreach (var track in (List<string>)data)
    {
        //Decode
        List(track);
        TypeConverter();

        Track newTrack = new Track(flightNum, x, y, altitude:alt, dateTimeNew);
        _tracks.Add(newTrack);
    }

    if (_tracks.Count == 0) //Break chain if no data
    {
        Console.Clear();
        Console.WriteLine(value:"No airplanes currently in airspace");
    }
    else
    {
        base.Handle(new List<Track>(_tracks));
    }
}
```

Figur 6 - Implementering af Handle() i Decoder

Her ses det at vi bryder kæden hvis der ikke er modtaget noget data, ellers sendes dataene videre i form af en liste af Tracks, ved at kalde base.Handle() som kalder Handle() i den abstrakte klasse Handler. Når alle led i kæden har implementeret Handle(), skal de kædes sammen, hvilket vi gør i vores ATMSystemFactory. Implementeringen af denne kan ses på Figur 7.

```
public class ATMSystemFactory
{
    private IHandler _receiver;
    private IHandler _decoder;
    private IHandler _regionFilter;
    private IHandler _trackUpdater;
    private IHandler _separationChecker;
    private IHandler _consolePrinter;
    private IHandler _logger;
    public void CreateATMSystem()
    {
        _receiver = new TransponderDataReceiver(TransponderReceiverFactory.CreateTransponderDataReceiver());
        _decoder = new Decoder();
        _regionFilter = new RegionFilter(new AirSpace(w:10000, e:90000, n:90000, s:10000, u:20000, l:500));
        _trackUpdater = new TrackUpdater();
        _separationChecker = new SeparationChecker(new SeparationCondition(verticalSeparationCondition:3000,horizontalSeparationCondition:10000));
        _consolePrinter = new ConsolePrinter();
        _logger = new Logger();

        //Chain Handlers
        _receiver.SetSuccessor(_decoder).SetSuccessor(_regionFilter).SetSuccessor(_trackUpdater)
            .SetSuccessor(_separationChecker).SetSuccessor(_consolePrinter).SetSuccessor(_logger);
    }
}
```

Figur 7 - Oprettelse af kæden

En demonstration af dette system kan ses i demonstrationsvideoen, med tilhørende forklaring.

4 Sammenligning med andre patterns

Cor gør brug af flere forskellige metoder, der kan findes i andre patterns og ligeledes overholder dette pattern også flere forskellige designprincipper.

I forhold til SOLID principperne overholder dette design pattern *Single Responsibility Principle* og *Open/Closed Principle*. SRP kommer til syne i de konkrete handler klasser, hvor hver har ét ansvar, der er indkapslet i klassen. Dvs. at hver handler klasse i kæden foretager én opgave/modificering på det data klassen får som input parameter, og derefter sendes det modificerede data videre til den næste handler klasse.

Denne kæde/linked liste struktur medfører også at det er nemt at indsætte nye led i kæden og ny funktionalitet, uden at modificere eller ødelægge allerede eksisterende kode. Hvis man eksempelvis vil implementere ny funktionalitet i systemet, laves der blot en ny handler klasse, som indsættes i denne kæde. Det er altså ikke nødvendigt at ændre i allerede eksisterende kode for at indsætte klassen/modulet, og ligeledes ødelægges der heller ikke andet kode ved at man indsætter et nyt led.

4.1 Observer patter

I observer pattern subscriber observers på et subject. Subjected giver besked til de observers, der subscriber på et givent event, om at en tilstand har ændret sig. I Cor kan der ses en variant af den funktionalitet der findes i Observer pattern, med at alle observers får besked om en tilstandsændring. I Cor kan det ses i det at hver konkret handler klasse/led i kæden får besked når der modtages data. Dvs. når den første i kæden kalder ”handle()” på basis klassen, så ryger dataen videre i kæden til de næste handlers. Denne funktionalitet fungerer lidt som subscriber funktionaliteten i observer pattern.

Dvs. at når en client indsætter en klasse/modul i kæden, så modtager den data fra klassen forinden. I forbindelse med Observer pattern ville dette være subscribe funktionaliteten, da klassen nu er indsat i kæden og dermed modtager data. Ligeledes kan modulet igen fjernes og på den måde unsubscribe i henhold til observer pattern.

4.2 Mediator

Dette pattern definerer et objekt der indkapsler hvordan andre klasser skal snakke sammen. Dvs. at der er meget lav kobling blandt klasserne, da klasserne indirekte snakker sammen gennem mediatoren. Så der er ikke nogen direkte kommunikation blandt klasserne, men alt foregår gennem mediatoren, der redirecter kaldene til den rette klasse.

Den samme funktionalitet kan ses i Chain of responsibility. Her sender hvert led i kæden også data videre til det næste led i kæden, men uden direkte at kommunikere med den næste klasse. Derimod kommunikerer hvert led i kæden med hinanden ”indirekte.” Den abstrakte basis handler klasse kan siges at være mediatoren i chain of responsibility, da det er denne klasse der kalder handle på det næste led i kæden, og sender dataene videre til den næste klasse.

Så hvert led i kæden kan siges at snakke sammen og sende data videre indirekte igennem basisklassen. Der blev også i den tidligere version af ATM Systemet anvendt en form for mediator pattern, da det var ATM Systemet der var ansvarlig for at sende data mellem de forskellige moduler og ”mediere” kontakten. Dette system var dog meget svært at teste da der lå en masse ansvar i ATMSystem klassen som var vores mediator. Når COR patternet anvendes er vores system blevet væsentlig nemmere at teste.

Dette pattern definerer et objekt der indkapsler, hvordan andre klasser skal snakke sammen. Dvs. at der er meget lav kobling blandt klasserne, da klasserne indirekte snakker sammen gennem mediatoren. Der er ikke nogen direkte kommunikation blandt klasserne, men alt foregår gennem mediatoren, der redirecter kaldene til den rette klasse. Den samme funktionalitet kan ses i Cor. Her sender hvert led i kæden også data videre til det næste led i kæden, men uden direkte at kommunikere med den næste klasse. Derimod kommunikerer hvert led i kæden med hinanden indirekte. Den abstrakte basis handler klasse kan siges at være mediatoren i Cor, da det er denne klasse der kalder handle på det næste led i kæden, og sender dataene videre til den næste klasse. Så hvert led i kæden kan siges at snakke sammen og sende data videre indirekte igennem basisklassen. Der blev også i den tidligere version af ATM Systemet anvendt en form for mediator pattern, da det var ATM Systemet der var ansvarlig for at sende data mellem de forskellige moduler og ”mediere” kontakten. Dette

system var dog meget svær at teste da der lå en masse ansvar i ATMSystem klassen som var vores mediator. Når COR patternet anvendes er vores system blevet væsentlig nemmere at teste.

5 Konklusion

Det kan konkluderes at Cor er meget brugbart i applikationer der skal processere noget data og i applikationer, hvor der er et data flow gennem systemet. Det er også meget nyttigt at kunne "stoppe" flowet i nogle situationer, ved at bryde kæden. Hvis man f.eks. validerer noget data i starten af kæden, og det viser sig, at denne data ikke er valid, kan man stoppe flowet således, at data ikke sendes videre i systemet. Det sparer CPU-tid og sørger også for, at unødvendige data ikke sendes videre i systemet. Cor giver også rigtig god mening ifht. OCP og SRP, da hver klasse har ét ansvar og det er nemt at tilføje ny funktionalitet uden at ændre i eksisterende kode. Derfor vil det generelt være smart at bruge Cor i applikationer hvor der er et flow af data, der filtreres mere og mere – og hvor der er mulighed for at bryde dette flow på et givent tidspunkt. Det vil også være smart i applikationer, hvor det skal være nemt at indsætte og fjerne nye moduler/processer, der skal modificere noget data på en bestemt måde. COR patternet bør ikke anvendes, når der altid kun vil være 1 specifik Handler, som vil håndtere requestet, eller hvis Clienten ved, hvilken Handler der børe håndtere requestet. Her vil de led i kæden som alligevel aldrig håndterer dataene, være overflødige og spild af CPU-tid.