# SW4SWD

*Obligatorisk Handin 1*

***Gruppe 25:***


| Navne | Studienummer |
|---|---|
| Fardoos Kelival | 202207724 |
| Bassam Salou | 202204580 |
| Hanad Hashi Osman Aden | 202204919 |
| Anthony Nguyen | 202109791 |


# Indhold

1 Indledning ..... 3
2 Beskrivelse af Chain of Responsibility Pattern ..... 4
2.1 Formål og type ..... 4
2.2 Struktur ..... 4
2.3 Dynamik ..... 5
2.4 Konsekvenser ..... 5
3 Diagrammer ..... 6
3.1 Klassediagram ..... 6
3.2 Sekvensdiagram ..... 7
4 Sammenligning med andre Design Patterns ..... 8
4.1 Sammenligning med Command Design Pattern ..... 8
4.2 Sammenligning med Observer Design Pattern ..... 8
4.3 Sammenligning med Iterator Design Pattern ..... 8
5 Eksempel og Implementering i C# ..... 9
5.1 ProductBaseHandler ..... 9
5.2 DistributeHandler ..... 10
5.3 CommodityHandler ..... 11
6 Konklusion ..... 12

# Chain-of-Responsibility

Automated Trading Bot

## 1 Indledning

Vi vil i denne rapport undersøge et design pattern, som endnu ikke er dækket i kurset for Software Design. I den forbindelse har vi valgt ”Chain of Responsibility” (som vi fremadrettet i rapporten vil forkorte til CoR), hvor vi vil give en fyldestgørende redegørelse for, hvilket design pattern vi har med at gøre samt give en analyserende refleksion over, hvordan vi har brugt CoR som design pattern og hvordan den relaterer sig til øvrige design patterns.

Vores overordnet projekt, som vi vil udvikle gennem CoR design pattern, er et system, vi har kaldt ”Automated Trading Bot”, som i essensen er et produkts-analyseprogram, hvor vi ud fra en klients forespørgsel omkring et produkt vil kunne analysere det og vurdere, om produktet er værd at investere i eller ej. Analysen af produktet afhænger af produktets type, eftersom der vil være forskel på om det er en aktie (stock), valuta (currency, crypto, forex) eller en råvare (Commodity) - og det er her vores design pattern kommer i spil.

Vi vil altså starte med at beskrive det valgte designmønster, herunder dets formål, type, struktur, dynamik og konsekvenser. Beskrivelsen understøttes af diagrammer, hvorefter vi vil foretage en sammenligning mellem CoR og andre relaterede mønstre for at klarlægge dets unikke egenskaber og anvendelsesområder. Endvidere vil vi præsentere vores C# implementering af CoR aka vores Automated Trading Bot for at demonstrere dets praktiske anvendelse. Og til sidst vil vi konkludere på vores undersøgelse ved at diskutere, hvornår mønsteret er nyttigt, og hvornår det måske ikke er hensigtsmæssigt at anvende det.

# 2 Beskrivelse af Chain of Responsibility Pattern

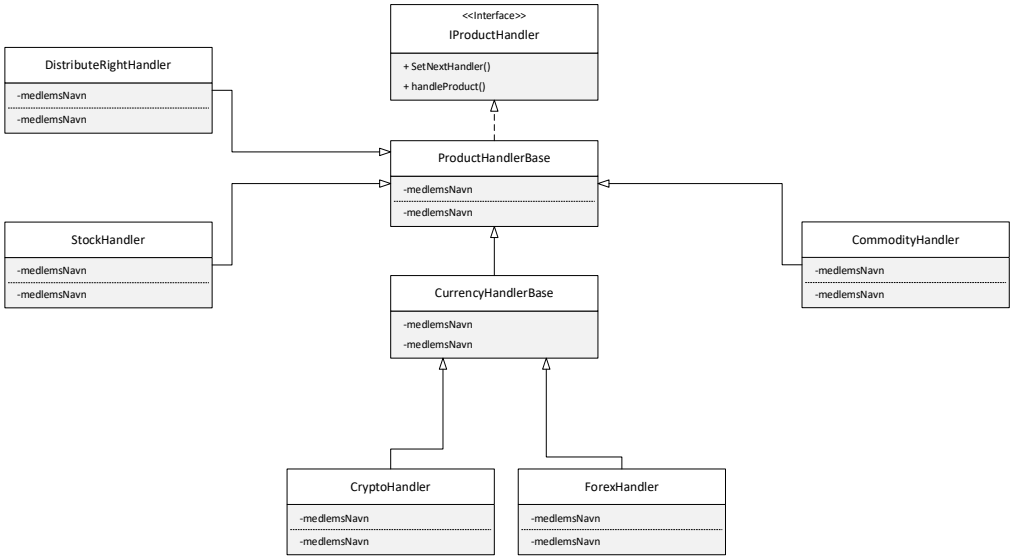
I dette afsnit vil vi dykke ned i detaljerne omkring CoR, herunder dets formål, struktur og dynamik.

## 2.1 Formål og type

CoR er et behavioural design pattern, som har til formål at etablere en løs kobling mellem afsendere af forespørgsler (klienter) og modtagere (handlers) ved at give mere end ét handler-objekt mulighed for at håndtere en forespørgsel. Ved at skabe en kæde af potentielle handlers sørger mønstret for at muliggøre behandling af en forespørgsel gennem kæden, indtil en passende handler er fundet. Dette pattern er velegnet til situationer, hvor der er behov for at håndtere forespørgsler hierarkisk og dynamisk samt ved variation i antallet eller typerne af handlinger, der skal håndtere en forespørgsel. Det bruges ofte i scenarier som eventhåndtering, filtrering af inputdata og behandling af anmodninger i webapplikationer.

## 2.2 Struktur

Strukturen af CoR Designpattern illustreres i diagrammet forneden:




*Figur 1: CoR klassediagram (Skitse)*

Figur 1 er blot en skitse af den arkitektur vi vil gøre brug af til implementeringen af vores CoR Pattern, men det der typisk kendetegner CoR strukturmæssigt er 4 hovedelementer:

- InterfaceHandler (IProductHandler): Deklarere en interface, som er fælles for alle handlers. I vores tilfælde har vi en interface, som har 2 metoder - 1 til at håndtere forespørgslen og en til at videregive forespørgslen til en anden handler.
- BaseHandler (ProductHandlerBase): En valgfri klasse som laver en general implementering af nogle funktioner, som alle handlerklasser gør brug af.
- ConcreteHandlers (StockHandler, CurrencyHandler, CommodityHandler): Indeholder handler-specifik kode til at håndtere en forespørgsel. Her har hver handler hver sine kriterier for, om en forespørgsel skal håndteres eller ej samt hvordan der passeres til næste handler.
- Client: Denne er ikke angivet i diagrammet, men det svarer til vores program.cs, hvori vi laver forespørgslen. Her kan man også gennem Clienten opsætte den næste handler, hvis det skulle være behov for dette.

## 2.3 Dynamik

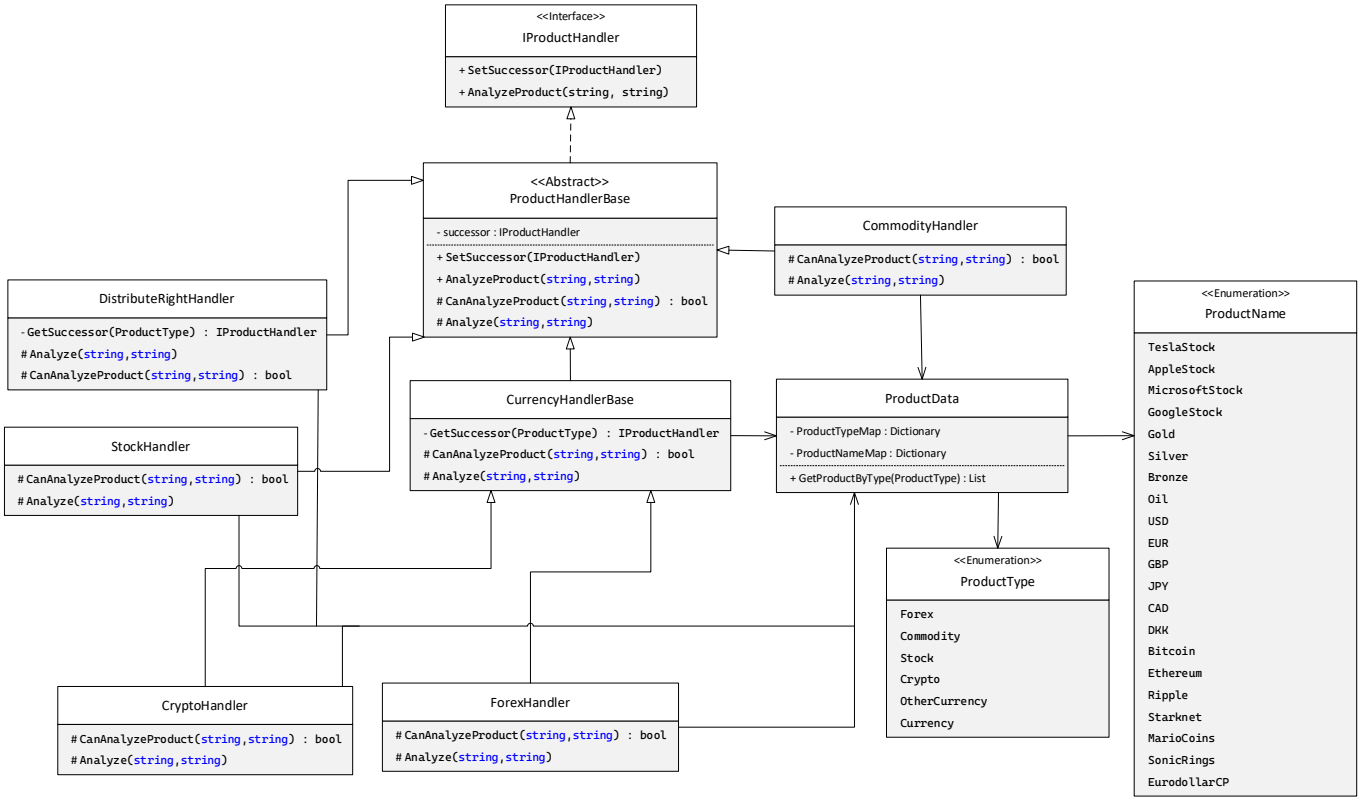
Dynamikken i pattern manifesteres i når en forespørgsel bevæger sig gennem kæden af handlers. Hver handler i kæden har mulighed for at håndtere eller ignorere forespørgslen, og hvis forespørgslen ignoreres vil handleren være i stand til at vælge dens ”successor” på en simpel måde. Alle handlers har nemlig hver deres ansvarsområde, hver deres ”kompetencer” og er i princippet uafhængige af hinanden. De kan fungere alene, hvilket også bidrager til muligheden for en dynamisk kædekonstruktion, hvor man altid kan ændre rækkefølgen af handlers. Dette skaber en fleksibel og udvidelig struktur, hvor nye handlers nemt kan tilføjes eller eksisterende handlers ændres uden at påvirke den overordnede funktionalitet.

## 2.4 Konsekvenser

De overordnede konsekvenser som CoR design pattern medfører er som nævnt den fleksibilitet og udvidelige struktur, hvor nye handlers nemt kan tilføjes eller eksisterende handlers kan ændres uden at påvirke den overordnede funktionalitet. Derudover er det værd at nævne, at strukturen for implementeringen af de enkelte handlers er genanvendelig, hvilket bl.a. ligger i deres arv-relation. Dette giver mulighed for at implementere et komplet system tidseffektivt.

# 3 Diagrammer

## 3.1 Klassediagram

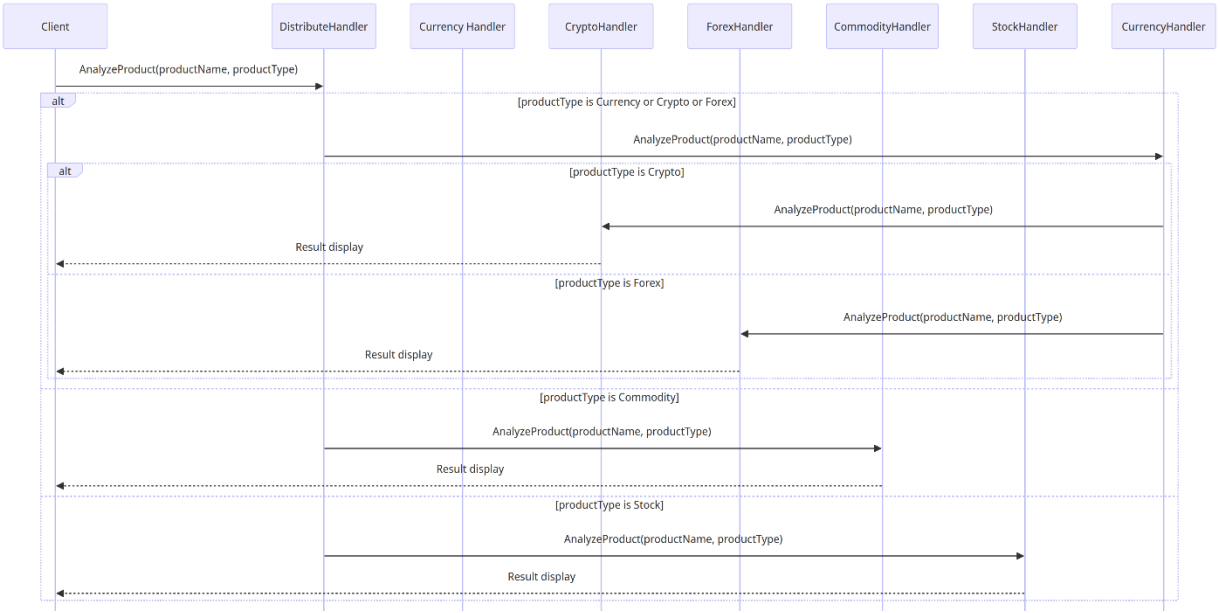



*Figur 2: Udvidet klassediagram*

Figur 2 illustrerer det udvidede softwaredesign, klassediagram som vi har gjort brug af til implementering af vores system. Vi starter med den generelle interface IProductHandler, som deklarerer 2 metoder, SetSuccessor og AnalyzeProduct. Disse realiseres i vores abstrakte base klasse, ProductHandlerBase, hvor de metoder fra interface bliver implementeret samt instantieret 2 abstrakte metoder, som hver concrete-handler skal implementere gennem override - CanAnalyzeProduct() og Analyze(). Alle concrete-handlers nedarver fra ProductHandlerBase, og kædekonstruktionen starter fra DistributeHandler, hvorefter den kan gå 3 veje: StockHandler, CurrencyHandler eller CommodityHandler. Dette dykker vi længere ned i sekvensdiagrammet og implementeringen.  Alle handlers gør brug af ProductData klassen, som blot er en mapping af enums. Det er gennem denne at handlers kan sætte kriterierne for om de kan håndtere forespørgslen samt hvordan de håndterer den.

## 3.2 Sekvensdiagram

På nedenstående figur ses vores sekvensdiagram, som håndterer de forskellige klasser for vores designmønster.




*Figur 3: Sekvensdiagram*

Systemet startes, når klienten initierer processen ved at sende en "AnalyzeProduct"-forespørgsel til DistributeHandler med et produktnavn samt produkttype. Derfra vil DistributeHandler validere produktet samt henvise til den relevante concretehandler baseret på produkttypen. Det er her, hvor vi har flere alternative cases, hvori et af casene har andre yderligere alternativer. DistributeHandlr kan passere forespørgslen videre til enten CurrencyHandler, StockHandler eller CommodityHandler. Hvis den passerer videre til CurrencyHandler vil denne enten færdiggøre analysen af produktet eller passere forespørgslen videre til enten CryptoHandler eller ForexHandler. Et af de mange handlers vil behandle forespørgslen og sende resultat tilbage til clienten. Sekvensdiagrammet illustrerer altså hvordan en anmodning kan køre gennem flere handlers, indtil den bliver behandlet.

# 4 Sammenligning med andre Design Patterns

CoR designmønstret deler visse ligheder med andre mønstre såsom Command, Observer og iterator, men adskiller sig også på flere måder.

## 4.1 Sammenligning med Command Design Pattern

I Command design pattern adskilles forespørgslen fra dens modtager ved hjælp af en kommandoobjekt. Dette objekt indeholder alle de nødvendige oplysninger for at udføre en bestemt handling, og det kan udføres på et senere tidspunkt, uden at afsenderen skal vide noget om modtageren. Derimod fokuserer CoR på at overføre ansvaret for håndtering af en forespørgsel gennem en kæde af modtagere/handlers. Mens Command-pattern giver mere fleksibilitet med hensyn til at udføre handlinger asynkront og i forskellige kontekster, giver CoR-mønstret mere fleksibilitet med hensyn til at tilføje eller fjerne modtagere dynamisk.

## 4.2 Sammenligning med Observer Design Pattern

I Observer design pattern etableres der en en-til-mange afhængighed mellem objekter, hvor et objekt (kendt som subjektet) holder styr på en liste over sine afhængigheder (kendt som observatørerne) og underretter dem automatisk om eventuelle ændringer i tilstanden. Mens dette pattern fokuserer på at opretholde konsistens mellem objekterne, fokuserer CoR på at delegere ansvaret for at håndtere forespørgsler gennem en kæde af modtagere. Observer-pattern er mere egnet til situationer, hvor der er en-til-mange relationer mellem objekter og en centraliseret kontrol af tilstanden, mens CoR er mere egnet til situationer, hvor der er behov for at distribuere ansvaret for håndtering af forespørgsler dynamisk.

## 4.3 Sammenligning med Iterator Design Pattern

Iterator design pattern fokuserer på at tilbyde en standardiseret måde at traversere elementer i en samling uden at afsløre dens interne struktur. Det der adskiller sig fra vores valgte pattern (CoR) er at det adresserer behovet for iteration og navigering i en samling, mens vores valgte design pattern fokuserer på at opretholde en kæde af objekter, der kan håndtere anmodninger. Iterator sikrer en løs kobling mellem klienten og samlingen ved at adskille traversal-logikken fra selve samlingen. Dette gør det muligt at anvende fleksibel og genbrugbar kode, hvorimod CoR letter anmodningshåndtering gennem en dynamisk kæde af ansvarlige objekter.

# 5 Eksempel og Implementering i C#

Nu hvor vi har diskuteret formålet og strukturen af CoR design pattern og sammenlignet det med andre relaterede patterns, vil vi illustrere implementeringen af dette design pattern. Denne implementering vil indeholde en kode for et system, der kan analysere forskellige produkter i form af aktier, valutaer (krypto, forex) og råvarer (heri. guld, sølv, bronze osv.). Dette system kan heraf bruges til at analysere og vurdere hvert produkt for at afgøre om det udvalgte produkt er værd at investere i. Dette bliver muliggjort ved brug af adskillige handler-klasser.

I de følgende afsnit vil vi gå igennem 3 af de vigtigste og centrale handler-klasser for illustrerer deres funktionalitet.

## 5.1 ProductBaseHandler

Som beskrevet under strukturen for CoR har vi en abstract-base class (basehandler), som inderholder den generelle ”template” implementering af vores handlers. Det er denne klasse, som alle conrete-handlers nedarver fra:

```
// Base class for product handlers
4 references
public abstract class ProductHandlerBase : IProductHandler
{
    protected IProductHandler successor;

    4 references
    public void SetSuccessor(IProductHandler successor)
    {
        this.successor = successor;
    }

    6 references
    public virtual void AnalyzeProduct(string productName, string productType)
    {
        // If this handler can analyze the product, do the analysis
        if (CanAnalyzeProduct(productName, productType))
        {
            Analyze(productName, productType);
        }
        // If there is a next handler, pass the product to it
        else if (successor != null)
        {
            successor.AnalyzeProduct(productName, productType);
        }
        // No handler in the chain can analyze the product
        else
        {
            Console.WriteLine("Error: Product not supported.");
        }
    }

    7 references
    protected abstract bool CanAnalyzeProduct(string productName, string productType);
    7 references
    protected abstract void Analyze(string productName, string productType);
}
```

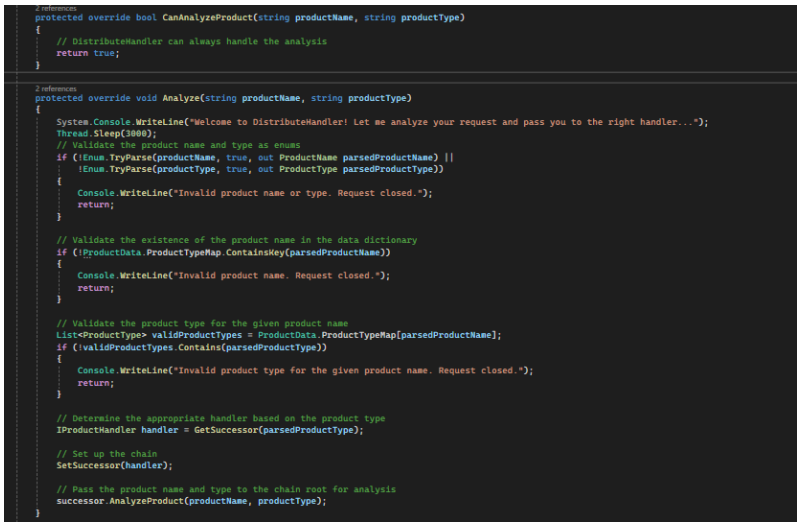
*Figur 4: ProductHandlerBase implementering*

Her implementeres SetSuccessor() og AnalyzeProduct() da disse metoder er tilbøjelige til at gå igen, mens CanAnalyzeProduct() og Analyze() med højest sandsynlighed skal overrides så de bliver handler-specifikke.

## 5.2 DistributeHandler

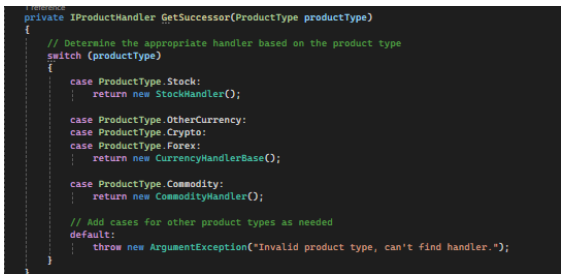
Distribute Handleren er en af de centrale elementer i implementering af systemet. Det fungerer som roden i vores kæde af ansvars-handler. Denne klasse er ansvarlig for at modtage anmodninger om produktanalyse og derefter dirigere dem til den relevante handlers baseret på produkttypen.

Her er koden til DistributeHandler-klassen, der er ansvarlig for håndtering af anmodninger om produktanalyse. Ved at udvide funktionaliteten gennem arv fra ProductHandlerBase-klassen får DistributeHandler mulighed for at håndtere forskellige typer af produkter.



```
2 references
protected override bool CanAnalyzeProduct(string productName, string productType)
{
    // DistributeHandler can always handle the analysis
    return true;
}

2 references
protected override void Analyze(string productName, string productType)
{
    System.Console.WriteLine("Welcome to DistributeHandler! Let me analyze your request and pass you to the right handler...");
    Thread.Sleep(3000);
    // Validate the product name and type as enums
    if (!Enum.TryParse(productName, true, out ProductName parsedProductName) ||
        !Enum.TryParse(productType, true, out ProductType parsedProductType))
    {
        Console.WriteLine("Invalid product name or type. Request closed.");
        return;
    }

    // Validate the existence of the product name in the data dictionary
    if (!ProductData.ProductTypeMap.ContainsKey(parsedProductName))
    {
        Console.WriteLine("Invalid product name. Request closed.");
        return;
    }

    // Validate the product type for the given product name
    List<ProductType> validProductTypes = ProductData.ProductTypeMap[parsedProductName];
    if (!validProductTypes.Contains(parsedProductType))
    {
        Console.WriteLine("Invalid product type for the given product name. Request closed.");
        return;
    }

    // Determine the appropriate handler based on the product type
    IProductHandler handler = GetSuccessor(parsedProductType);

    // Set up the chain
    SetSuccessor(handler);

    // Pass the product name and type to the chain root for analysis
    successor.AnalyzeProduct(productName, productType);
}
```

*Figur 5: DistributeHandler – kodeafsnit 1*



```
1 reference
private IProductHandler GetSuccessor(ProductType productType)
{
    // Determine the appropriate handler based on the product type
    switch (productType)
    {
        case ProductType.Stock:
            return new StockHandler();

        case ProductType.OtherCurrency:
        case ProductType.Crypto:
        case ProductType.Forex:
            return new CurrencyHandlerBase();

        case ProductType.Commodity:
            return new CommodityHandler();

        // Add cases for other product types as needed
        default:
            throw new ArgumentException("Invalid product type, can't find handler.");
    }
}
```

*Figur 6: DistributeHandler – kodeafsnit 2*

Det ses herover at Distribute-handler klassen implementerer følgende 3 metoder.

1. *CanAnalyzeProduct-metoden* er overskrevet for at indikere, at DistributeHandler altid kan håndtere analysen af et produkt. Dette skyldes, at DistributeHandler fungerer som roden i kæden og bør være i stand til at håndtere enhver anmodning, der når den.
2. *Analyze-metoden* er også overskrevet for at udføre den specifikke analyse af produktet, når det kommer til denne handler i kæden. Denne metode viser også en velkomstbesked og sender derefter anmodningen videre til den relevante handler i kæden ved hjælp af GetSuccessor-metoden.
3. *GetSuccessor-metoden* spiller en afgørende rolle ved at bestemme den næste handler i kæden baseret på produkttypen. Ved at bruge en switch-case-struktur undersøger metoden produkttypen og returnerer den korrekte handler, der skal analysere produktet. Dette sikrer, at anmodninger om produktanalyse bliver routet korrekt gennem kæden af concretehandlers baseret på deres specifikke produkttype.

Ved brug af disse metoder kan vi effektivt gøre det muligt at håndtere produktanmodninger gennem kæden af ansvars-handlers.

## 5.3 CommodityHandler

CommodityHandler-klassen er den klasse, der er ansvarlig for at analysere råvarer såsom guld, sølv, bronze og olie. Denne handler-klasse udgør en vigtig del af systemet ved at muliggøre en analyse af disse typer produkter. Dette sker ved at klassen arver også fra ProductHandlerBase-klassen for at kunne håndtere forskellige former for råvarer.
Nedenfor illustreres koden til CommodityHandler-klassen, der er ansvarlig for analysen af råvarer såsom guld, sølv, bronze og olie.

```
2 references
protected override bool CanAnalyzeProduct(string productName, string productType)
{

    // Check if the product type is "Commodity" and the product name exists in the ProductData
    return productType == "Commodity" && Enum.TryParse(productName, true, out ProductName productNameEnum) &&
           ProductData.ProductTypeMap.ContainsKey(productNameEnum);
}
```

*Figur 7: CommudityHandler – kodeafsnit 1*

```
protected override void Analyze(string productName, string productType)
{
    if (productName == "Gold")
    {
        Console.WriteLine("CommodityHandler: Analyzing Gold...");
        Thread.Sleep(3000);

        // Perform analysis and determine if worth buying
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Result: Gold is worth buying.");
        Console.ResetColor();
        return;
    }

    if (productName == "Silver")
    {
        Console.WriteLine("CommodityHandler: Analyzing Silver...");
        Thread.Sleep(3000);

        // Perform analysis and determine if worth buying
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Result: Silver is NOT worth buying.");
        Console.ResetColor();
        return;
    }

    if (productName == "Oil")
    {
        Console.WriteLine("CommodityHandler: Analyzing Oil...");
        Thread.Sleep(3000);

        // Perform analysis and determine if worth buying
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Result: Oil is worth buying.");
        Console.ResetColor();
        return;
    }

    if (productName == "Bronze")
    {
        Console.WriteLine("CommodityHandler: Analyzing Bronze...");
        Thread.Sleep(3000);

        // Perform analysis and determine if worth buying
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Result: Bronze is not worth buying.");
        Console.ResetColor();
        return;
    }
}
```

*Figur 8: CommudityHandler – kodeafsnit 1*

I CommodityHandler-klassen er de to abstrakte metoder, CanAnalyzeProduct og Analyze, blevet implementeret til at definere funktionaliteten for produktanalyse.

1. *CanAnalyzeProduct-metoden* sikrer, at CommodityHandler kun behandler anmodninger om analyse, der relaterer sig til råvarer, og den specificerer de præcise typer af råvarer, som denne klasse kan håndtere.
2. *Analyze-metoden* styrer selve analysen af produktet ved at vælge den korrekte handlingsproces baseret på produkttypen.

Gennem disse implementerede metoder kan nye produktbehandlingsklasser nemt oprettes.

# 6 Konklusion

I denne rapport har vi undersøgt Chain of Responsibility (CoR) design pattern og dets anvendelse i udviklingen af vores projekt, Automated Trading Bot. Formålet med CoR er at etablere en dynamisk kæde af handlers til håndtering af forespørgsler hierarkisk og fleksibelt. Vi har vist, hvordan CoR-pattern adskiller sig fra andre design patterns som Command, Observer og Iterator, og vi har præsenteret en dybdegående implementering af pattern i C#. Gennem vores C# implementation har vi vist, hvordan vi har oprettet en kæde af handlers, der kan analysere forskellige typer produkter, herunder aktier, valutaer, krypto og råvarer. Vi har fremhævet vigtige klasser som ProductBaseHandler, DistributeHandler og CommodityHandler og demonstreret deres rolle i analysen af produkter.

Efter udførelsen af alt dette er vi kommet frem til, at vores implementering har vist, at CoR-pattern giver en struktureret tilgang til at håndtere forskellige typer produkter. Designet har gjort det muligt at udvide systemet med nye produkttyper uden at påvirke den eksisterende funktionalitet.

Det er tydeligt, at CoR-pattern demonstrerer sin styrke mest effektivt i komplekse systemer, hvor der er behov for fleksibel, hierarkisk og dynamisk håndtering af forespørgsler. I sådanne miljøer muliggør CoR en elegant løsning, hvor forskellige handlers kan tilføjes, fjernes eller ændres uden at påvirke den overordnede funktionalitet i systemet. Dette er især nyttigt i situationer, hvor behandlingen af forespørgsler kan variere baseret på forskellige betingelser eller kontekster, hvilket kræver en fleksibel og udvidelig tilgang.

Når dette er sagt, kan CoR-pattern være også være velegnet i simple eller lineære systemer, hvor hierarkiet af ansvar er fastlagt på forhånd, da håndteringen af forespørgsler blot vil ske statisk. Det er i sådanne tilfælde at man blot kan konstrurere kæden i Client koden uden at skulle lave en indviklet logik til handler distribuering.

Samlet set har vores erfaring med CoR-mønsteret været positiv, og vi erkender dets værdi som et værktøj til effektiv og struktureret softwareudvikling.