Command Pattern

Gruppe 12

Brian Hansen 201310502

Jeppe Tinghøj Honoré 201371186

Toke Tobias Aaris 201407321

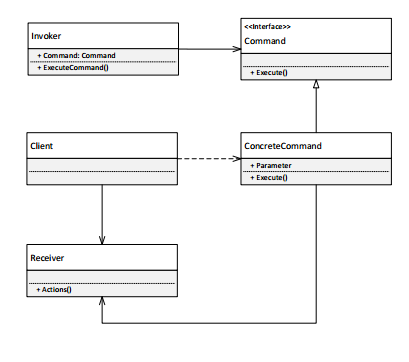
# Command pattern

Command pattern er et behavioral pattern, som muliggør at behandle kommandoer som objekter i stedet for metoder. Dette gøres ved at objektet indkapsler al information, der er brug for at udføre en ønsket handling. På denne måde kan der sendes kommandoer til objekter uden at der behøves kendskab til implementeringen eller modtageren af kommandoen.

I command-mønstret arbejder man med forskellige klassetyper, der hver især har en vigtig rolle:

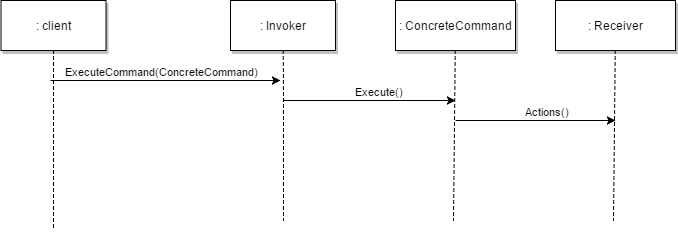
* Invoker
* Command
* Concrete Command
* Receiver
* Client

Invoker har ansvaret for at initiere eksekveringen af de kommandoer, der bliver sendt til den af klienten. Command er et interface, der har de pågældende generiske metoder, som Concrete Command implementerer og disse klasser repræsenterer de commands, der skal eksekveres. Receiver indeholder de konkrete metoder, der bliver eksekveret. Client er den klasse, der anvender mønstret, og det er her kommando-objekterne bliver instantieret og linket til receiver.



Figur - Klassediagram over command pattern

I klassediagrammet ses relationerne mellem klassetyperne i mønstret. Det er dog kommunikationen mellem disse, hvilket bedst beskrives ved et sekvensdiagram, som ses i figur 2. Her ses det, at klienten sender kommandoobjekter til invoker, som så sørger for at eksekverer metoderne på det pågældende objekt, som yderligere kender receiver og den konkrete implementering af algoritmen, som eksekveres.



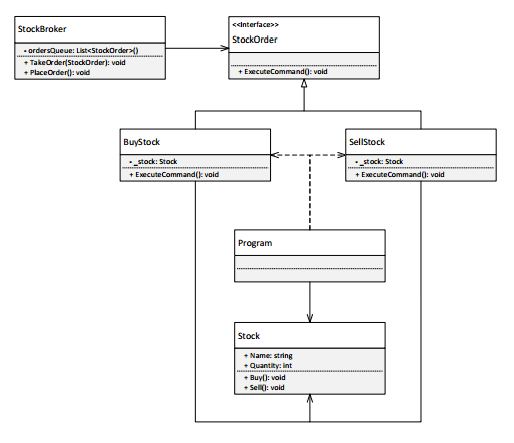
Figur - Sekvensdiagram over command pattern

Mønstret er set i mange afskygninger, og vi har valgt den model, der i vores øjne bedst forklarer konceptet. En variation er at lade interfacet være abstract og lade denne klasse håndtere forbindelsen til receiveren. For overskuelighedens skyld har vi placeret denne kode i de konkrete kommandoklasser.

## Eksempel

Et eksempel kan være et aktiesystem med forskellige aktører i form af klasser. Antag at klienten gerne vil sende kommandoer om enten at købe eller sælge en aktie til en børsmægler. Klienten er ligeglad med, hvordan implementeringen ser ud, og hvem der udfører den. Børsmægleren vil i dette tilfælde være invokeren, mens vi har to konkrete kommandoer – BuyStock og SellStock – som er dem, der ved, hvor implementeringen af koden ligger, og kalder disse metoder på Stock som er receiver.

Ved at implementere løsningen på denne måde, kan klienten tilføje forskellige kommandoer til en ordrekø i invokeren ved at kalde *TakeOrder()* og bede invokeren om at udføre ordrene ved at kalde *PlaceOrder()*.



Figur 3 - Klassediagram over Stock-program

## Anvendelighed

Et sted hvor mønstret er hyppigt anvendt er, hvis man ønsker at lave en form for kø over kommandoer. Ved implementeringen heraf, kan man fx lave en liste med kommando-objekter, der så eksekveres i en bestemt rækkefølge og efter behov.

Et konkret eksempel herpå er et programs mulighed for at lave en fortryd-kommando. Antag at vi arbejder med en lommeregner. Ved hver operation vil programmet skifte state. Typisk vil man i sådan et program gerne have muligheden for at gå tilbage til et tidligere state.

For at løse denne problemstilling kunne man anvende kommandoer og i Command-interfacet definere en *Do()*- og *Undo()*-metode. At plusse et tal med et andet her udgøre en Concrete Command, der implementerer interfacet. I *Do()*-metoden eksekveres algoritmen til at plusse to tal med hinanden, mens Undo()-metoden eksekverer den omvendte algoritme at trække de samme tal fra i hinanden i invoker, hvor metoderne er implementeret. Hvis *Do()-*metoden plusser to tal sammen, vil *Undo()*-metoden trække de samme tal fra hinanden. På denne måde kan programmets states simpelt og overskueligt ændres frem og tilbage med do og undo.

Fordelen ved at bruge kommandomønstret til problemstillinger som denne er, at det er meget vedligeholdelsesvenlighed og indeholder ingen overflødig information, samt at det ikke er hukommelseskrævende. Mønstret gør det nemt at tilføje ny funktionalitet til programmet, hvilket stemmer godt overens med open/closed-princippet og ved at hver kommando har sin egen klasse, overholdes princippet om single resposibility.

Anvendelsen af interface til at definere de konkrete kommando-klassers metoder giver mønstret et højt abstraktionsniveau og stemmer godt overens med interface segregation-princippet og Livskov substitutionsprincip samt dependency inversion-princippet. Så går man ind for SOLID, er der her tale om et dydsmønster.

Der ligger dog et stort arbejde i at implementere kommandoklasser med *Do()* og *Redo()* til samtlige af programmets operationer. Man risikerer at ende ud med en masse små konkrete klasser, der kan ødelægge overblikket af designet.

Typisk vil mønstrets anvendelighed først rigtigt komme til sin ret i større systemer, da det nemt kan få små og simple programmer til at fremstå mere komplekse, end de er, og benefit vil være lille.

# Sammenlignelige patterns

## Strategy

Der er en del ligheder mellem command pattern og strategy. Først og fremmest har de begge til formål at indkapsle implementeringsinformation i forskellige klasser og bruger et interface som forbindelse her i mellem. Den store forskel ligger dog i, at selve handlingen gøres til et objekt i command mønstret, mens metoden i strategy blot kan kaldes direkte fra klienten. Også anvendelsen er typisk forskellig, da strategy ofte sammenkobler to metoder, der overordnet set har samme opgave, som skal løses på forskellige måder. Det kunne fx være at skrive til en json-fil eller en xml-fil og indkapsle de forskellige algoritmer i forskellige klasser.

## Memento

Memento mønsteret er som Command et mønster der simulerer en bestemt adfærd. Denne adfærd kan eksempelvis beskrives med fortryd- / gendan-funktioner i et billedbehandlingsprogram. Memento gemmer et objekts state i eks. En liste. Denne liste kan så benyttes til at springe tilbage til objektets tidligere tilstand, ligeledes kan der springes frem hvis der findes objekter frem i listen.

Command er ikke specielt anvendeligt til billedbehandlingsprogrammet, fordi det har en liste over de kommandoer der er kørt, hvis en handling fortrydes skal der benyttes meget processorkraft til at tegne om på billedobjektet. Ligeledes er der funktioner der er destruktive for billedet eks. Blur, ændre opløsning eller slet. Disse ting kan man ikke fortryde med en Command, da man ikke kan gendanne et billedobjekts indhold med en Command uden information om den forrige tilstand for billedobjektet. Desuden vil det generelt være processortungt at køre rent Command mønster til at køre modsatrettede-kommandoer på billedobjekter, hvis det overhovedet kan lade sig gøre.

Memento kan benyttes til at holde information omkring billedobjektets tidligere tilstand. Dette objekt kan gemmes med kommandoen i Command-stacken, så når en kommando poppes af stacken for at rulle tilbage vil Memento-token findes med kommandoen. Denne måde at gendanne billedets tilstand vil kræve mere hukommelse da billedobjektet gemmes hver gang en kommando køres.

Så i det specifikke tilfælde med et billedbehandlingsprogram, vil Command mønsteret ikke kunne stå alene hvis en fortryd- / gendan-funktion skal implementeres. Nedenfor er et pseudokode eksempel, der viser hvilke problemer man får, når man vælger et Command-mønster alene uden Memento, samt en implementation med Memento.

### Memento kodeeksempel uden Command

//Pseudocode example command without memento

public interface Command

{

void Execute();

void Undo();

}

class BlurCommand : Command //Destructive command, cant undo

{

private Picture picture;

public BlurCommand(Picture pic)

{

this.picture = pic;

}

public override void Execute()

{

picture.BlurPicture();

}

public override void Undo()

{

picture.UnBlur();//Can't be done, info about original picture lost

}

Figur 4 - Pseudokode, Command i billedbehandlingsprogram

### Memento kodeeksempel med Command

//Pseudocode example command with memento

public interface Command

{

void Execute();

void Undo();

}

class BlurCommand : Command//Destructive command, can’t undo with command

{

private Picture picture;

private Picture previousPicture;

public BlurCommand(Picture pic)

{

this.picture = pic;

previousPicture = this.picture.Pixels;//Save Memento token

}

public override void Execute()

{

picture.BlurPicture();

}

public override void Undo()

{

//Rollback, with Memento. Restore to object to previous state

picture = previewPicture;

}

}

Figure 5 – Pseudokode, Command med Memento-token

# Konklusion

Command pattern er et adfærdsmønster, der muliggør indkapsling af metoder som objekter. Dette giver grobund for fx at kunne lave køer af kommandoer og kan være en af byggestenene i en do/undo-funktionalitet. Mønstret følger en lang række af principperne, vi kender fra SOLID, og det er derved et mønster, der er nemt at vedligeholde. Dog skal det også give mening at bruge, da der kan forekomme en hel del arbejde i at implementere kommandoklasser til samtlige operationer, og det kan have den effekt, at softwaredesignet fremstår komplekst. Det bør derfor overvejes nøje, om anvendelsen giver mening og gør mere gavn en skade.

# Referencer

<http://www.blackwasp.co.uk/gofpatterns.aspx>

<https://www.tutorialspoint.com/design_pattern/command_pattern.htm>

<https://sourcemaking.com/design_patterns/command>

<https://www.youtube.com/watch?v=jOnxYT8Iaoo>

<http://www.parkeresmay.com/memento-and-command-pattern-for-undo/>

<https://matt.berther.io/2004/09/16/using-the-command-pattern-for-undo-functionality/>