SWD Design Patterns
Gruppe 38

Aarhus University School of Engineering

CQRS Architectural Pattern

12-04-2018

Auld	Studienummer	Navn
au557919	201607110	Kasper Juul Hermansen
au516160	201400298	Karsten Winther Johansen
au559220	201606374	Jakob Levisen Kvistgaard
au339657	20112806	Martin Lynge Dalgaard
au567214	201607589	Jakob Bonde Nielsen

Antal karakterer: NumberOfCharacters

Indholdsfortegnelse

1 Introduktion

I denne opgave skal vi undersøge et Design Pattern - som ikke har været gennemgået på kurset. På baggrund af interesse har vi valgt **CQRS**, som er et Architectural Pattern. Ud fra opgaven, har vi undersøgt hvordan **CQRS** fungerer, og hvad der kræves for at for at implementere det, for at opfylde kravene.
Dette er dokumenteret med denne rapport, en præsentationsvideo, et powerpoint slide, en Visual Studio Solution som beskriver CQRS & CQRS/Event sourcing og en video demonstration af ovenstående.

1.1 Arkitektonisk vs Design

Generelt er ideen med at bruge et pattern, en idé til at løse et problem bedst muligt. Det kan være alt fra store komplekse enterprise systemer, ned til små løsninger, som skal effektiviseres. Ideen med dem alle er at få bedre overblik over løsningen, samt at løse disse problemer på en genkendelig måde.

Design patterns, bruges til at løse et gentagende problem, som kan opstå. De er derfor generelt problemløsende for udviklere, da de afhjælper med at løse komplekse problemer. Der er mange fordele ved at bruge et sådant pattern, typisk til at lave en bedre løsning for et problem, i praktisk vil det typisk ende med en renere kodestil, som er nemmere at sætte sig ind i og forstå. Derfor er mange af disse design patterns blevet generelle at bruge, de de har vist deres fordele, når man begynder at bruge dem.

Generelt vil både Design og Arkitektoniske patterns give en ide til hvordan løsningen er, men hvor et design pattern har mere fokus på lokale problemer, vil et arkitektonisk pattern have et mere abstrakt oveblik over systemet, og ikke en nær så konkret løsning. Et arkitektonisk pattern giver et overblik over system strukturen, typisk med software komponenter, og hvordan de snakker sammen. Et arkitektonisk pattern har derfor primært fokus på aspekter, som ydeenve, skalerbarhed, pålidelighed, testbarhed, vedligeholdese muligheder og mange flere faktorer. Derfor bliver disse designs typisk brugt i større komplekse situationer, hvor f.esk arbejdsbyrden skal deles op, eller visse regler skal følges.

Overordnet kan det forståes at Design patterns fokusere på en lokalt problem, som indgår i en overordnet struktur. Arkitektoniske patterns fokusere derimod på et mere overordnet problem, som definere en mere væsentlig del af strukturen, og kan sætte en standart for den.

1.2 Termliste

CQRS - Command Query Responsibility Segregation
CQS - Command Query Separation
CRUD - Create Read Update Delete
Design Pattern Design mønster - en til flere klasser der beskriver et generalt problem, via. en predifineret løsning.
Architectural Pattern Arkitektonisk mønster - en general struktur på en overordnet problem, som har stor betydning for struktur og funktionalitet.

CQRS 2

CQRS står for **C**ommand **Q**uery **R**esponsibility **S**egregation, og er et arkitektonisk mønster (Architectural Pattern), som går ud på at optimere den måde at skrive og læse til et domæne på. Pattern'ets kerne er, at kunne individualisere det at læse og skrive til et domæne, samt at give mulighed for et koncept der hedder Event Sourcing, som vil blive beskrevet senere i rapporten. Pattern'et giver mulighed for, at skalere det at læse og skrive helt seperat, så man kan bruge de resourcer der er til rådighed, helt naturligt.

2.1 Brug

Dette pattern bruges ligesom **CRUD**, til at skrive til og læse fra en Database. Forskelligt fra **CRUD** er læse og skrive operationerne separeret fra hinanden. Det betyder, at dette pattern er velegnet til at indgå i følgende scenarier.

- Applikationer, hvor flere læse- og skriveoperationer foregår parallelt.
- Applikationer, hvor der er behov for skalering
- Applikationer, hvor skrive- og læselogik skal udvikles uafhængigt
- Applikationer, hvor kompleksiteten overstiger en simpel CRUD implementering
- Applikationer, hvor domænet og forretningslogikken overstiger en vis kompleksitet

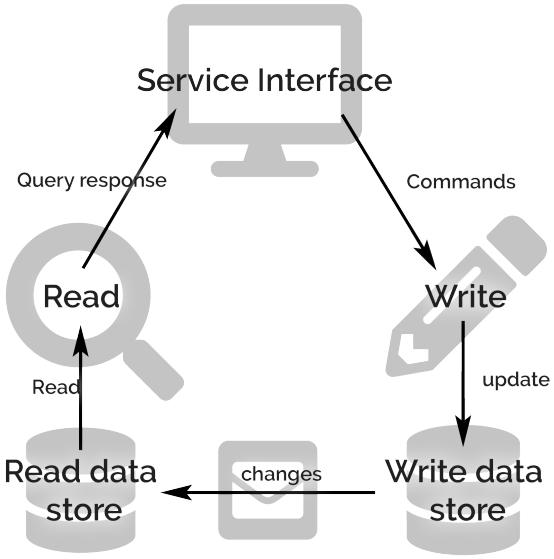
Grunden til, at **CQRS** er velegnet til disse anvendelser er, at kommandoerne kan udvikles med en grad af tilpassethed, hvilket betyder at problemer i forbindelse med sammenfletning af data kan formindskes betydeligt. Ved brug af **CQRS**, og derved separeret udvikling af læse/skriveoperationer. Er det også muligt, at lade mere af forretningslogikken indgå i skrive operationerne, uden at dette påvirker læsningen.

CQRS er særdeles anvendeligt ved skalering af komplekse systemer, da det ofte er gældende, at der er uligevægt i antallet læse- og skriveoperationer. Det medvirker til, at den separerede udvikling af disse er eftertragtet.

2.2 Type

CQRS er af typen Architetural Pattern, hvilket beskriver et bredere omfang end Design Patterns. Det vil sige at CQRS kan beskrive en general forståelse af et system, hvor et design pattern kan bruges til at håndtere et mindre problem. CQRS stammer fra CQS (Command-Query Seperation) af Bertrand Meyer (Wikipedia 2014). Dette Pattern tilhører den klassifikation af Architetural patterns, der bruges til Read og Write, som f.eks. CRUD også implementere.

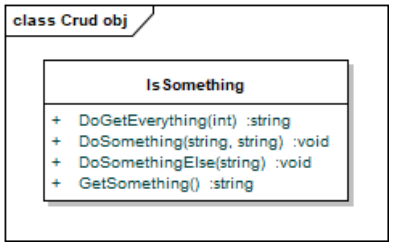
2.3 Struktur



Figur 2.1: Simpel CQRS med Read og Write

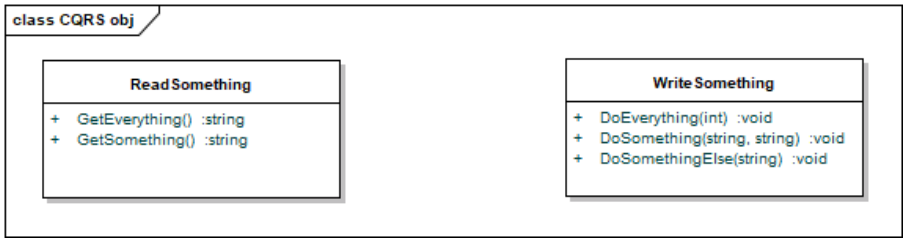
Dette pattern består af 2 vitale dele, Read siden og Write siden, Disse 2 dele, gør at patternet effektivt kan skrive eller læse til en data store (domæne). Princippet er at disse dele er Rent (Pure) altså kun gør det som den er sat til. Write siden skriver kun til Datastore og returnere ikke nogen form for tilstand tilbage. Read siden ændre ikke data, men returnere kun Data. Disse handlinger er gjort igennem Commands og Queries.

2.3.1 Generelt



Figur 2.2: Crud klasse

Et normalt objekt indeholder både Read og Write operationer, og nogle gange kombineret, se figur 2.2. Denne klasse refactors til at implementere CQRS som set i 2.3



Figur 2.3: CQRS klasse

"CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query" - Greg Young

Generelt består CQRS af Commands (WriteSomething) og Query reponses (ReadSomething) samt Events, se figur 2.3. Commands rejser events når de bliver oprettet, og gør det muligt at opdatere Read siden. Denne opdeling gør det muligt, at udvikle disse sider totalt separeret, bortset fra den event bus, der binder det hele sammen. I nogle variationer kan der eksempelvis bruges være brugt en enkelt database som persisterings model og en anden variation bruger en til hver side af systemet. Begge løsninger har fordele og ulember, hvilket er et spørgsmål om individualisering og kompleksitet, hvor en enkel database / domæne er nemmere at implementere end 2 databaser og et ikke delt domæne.

Et scenarie der kan opstå er: Service interface sender en Command, til at ændre en model f.eks. oprette, slette, redigere en model, dette Command bliver valideret og bliver først persisteret på Write data store, men bliver også sendt til Read data store. Read data store bliver nu opdateret med frisk data, og nu opdateret med Write data store. I en anden instans kan Service interface anmode Read siden, med et query om en model fra Read data, og ønske at modtage et stykke data. Read siden opretter nu en DTO (Data Transfer Objects), og returnere den med det ønskede data.

2.3.2 Read

Read siden består Queries, hvor andre modeller kan anmode om DTO'er dette gør det let at hente data, næsten direkte fra Read data store, som næsten altid er opdateret af Write siden. Read data store, opfylder dog ikke *Consistency* princippet, men nærmere *Eventually Consistent* som gør at systemet skal tage højde for at der kan være samhørighedsproblemer på dataen.

2.3.3 Write

Write siden består af Commands som kan ændre de modeller som findes i domain modellerne. Write siden opdatere sin egen data store, samt Read data store.

2.4 Konsekvens

Umiddelbart virker **CQRS** som et simpelt pattern, men ved at implementere dette kan kompleksiteten af systemet hurtig øges, da dette oftes inkludere brugen af Event Sourcing, som implementeringsmæssigt kan forekomme ret omfattende. Grunden til, at Event sourcing ofte inkluderes er, et ønske om et system der kan følge op på hændelser. [1]

En anden konsekvens ved at benytte **CQRS** er, at man ved at separere læse- og skriveoperationer kan komme ud for, at den data der læses kan være i et stadie, der ikke stemmer overens med det forventede.

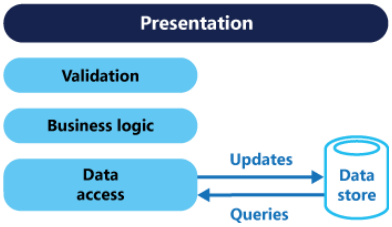
2.5 Andre patterns

Ved sammeligning af **CQRS** og andre patterns, er det nærliggende at inddrage **CRUD**.

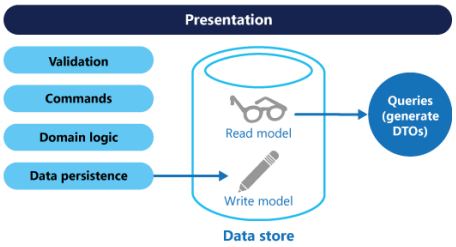
Begge disse patterns bruges til, at tilgå og skrive data på en Database, men deres metoder er forskellige. Forskellen udspiller sig i måden de er implementeret på, og derfor også måden de operere på.

På understående figur 2.4 nedenfor kan det ses, hvordan **CRUD**-implementeringen ved hjælp af sin *Data-access*-klasse, både tilgår og skriver data på databasen. Derimod er dette separeret på figur 2.5, dette fremgår også af figur 2.2 og 2.3

[1] FiXme Fatal: Tilføj eksempel for hændelse



Figur 2.4: **CRUD**-Implemantation (Microsoft 2017)



Figur 2.5: **CQRS**-Implemtation (Microsoft 2017)

Ved at separere læse og skrive operationerne, opnås skaleringsmuligheder der med **CRUD** ikke er mulige. Grunden til dette er individualiseringen af både læse og skrive klasser, samt de afhængigheder de ikke deler.

Derudover behøver disse ikke at udvikles 1:1.

Den største forskel på de to patterns udspiller sig i, at **CRUD** udvikles som regel med **DDD D**omain **D**riven **D**esign, hvorimod **CQRS** udvikles ofte i et begivenhedsdrevet design (Event Driven Design). Som navnene antyder, bliver det domæne drevne design bygget op af domæner, som figurer i den virkelige verden, hvorimod eventdriven design er bygget op om begivenheder der sker. F.eks. at en kunde bestiller en kop kaffe.
Ved til dels at opbygget et system om det eventdrevne princip, og at adskille læse/skrive-operationer mødes nogle problematikker som skal løses. Disse problemer gør, at **CQRS** kun bruges når systemet opnår en vis størrelse og kompleksitet, men at **CQRS** forstrækkes når dette niveau opnås.

Ved implementering af **CQRS** gøres der brug af Event Sourcing, som også vil blive omtalt herefter. I korte træk er det den del der sørger for, at fuldende den eventdrevne tankegang og underbygge reaktion indbyrdes i pattern'et. Det er dog nævneværdigt, at eventsourcing ikke er en del af selve **CQRS**-pattern'et men at det ofte bliver implementeret ved brug af **CQRS**, da man på den måde løser nogle af ovenstående problemer.

Event Sourcing 3

"We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there." - ***Martin Fowler*** *on Event Sourcing*

Event sourcing er et andet arkitekonisk pattern, som går ud på at gemme data i en sekvens af events, Event sourcing er som regel brugt i konjunktion med CQRS, da dette pattern tilbyder en naturlig adskildelse som Event sourcing, kan benytte til at give optimal effekt.

3.1 Hvorfor?

Event sourcing gør det muligt at gemme en historik af alle events af en type, dette er især optimalt, fordi data er så værdifuldt. Som ejer af en forretning ville det kun være optimalt, hvis man kendte alle de trin en kunde gik igennem, når man skulle modtage vedkommendes vare, end kun at vide om den kom frem, eller ikke kom frem. Denne historik giver ejeren en chance for, at finde ud af hvor tingene gik galt, og hvordan virksomheden skal udrette dette problem. Eventsourcing er iøvrigt brugt meget i store enterprise applikationer, hvor der er resourcer til at implementere dette pattern, samt at goderne kan udligne de negative ting ved det, som f.eks. kompleksistet, ydevne og omkostninger.

3.2 Brug

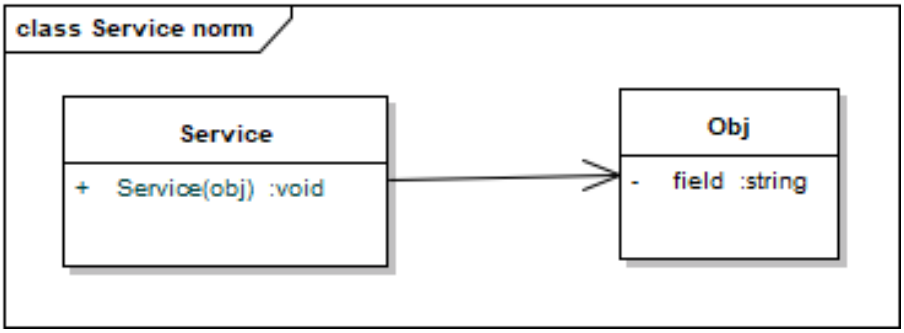
Event Sourcing er brugt i de applikationer hvor transaktionerne / historik er kritisk for forretnings modellen, på den applikation der har implementeret den. Event sourcing har nogle ulemper, det er kompliceret at implementere, styre og vedligeholde. For nogle applikationer kan det være nyttigt, at kunne se denne sekvens af handlinger, fordi det lige præcis er nødvendigt af overstående grunde.

I de følgende scenarier kunne det være nyttigt

- Applikationer, hvor historik er en prioritet.
- Applikationer, hvor handlinger skal være uforanderligt (immutable).

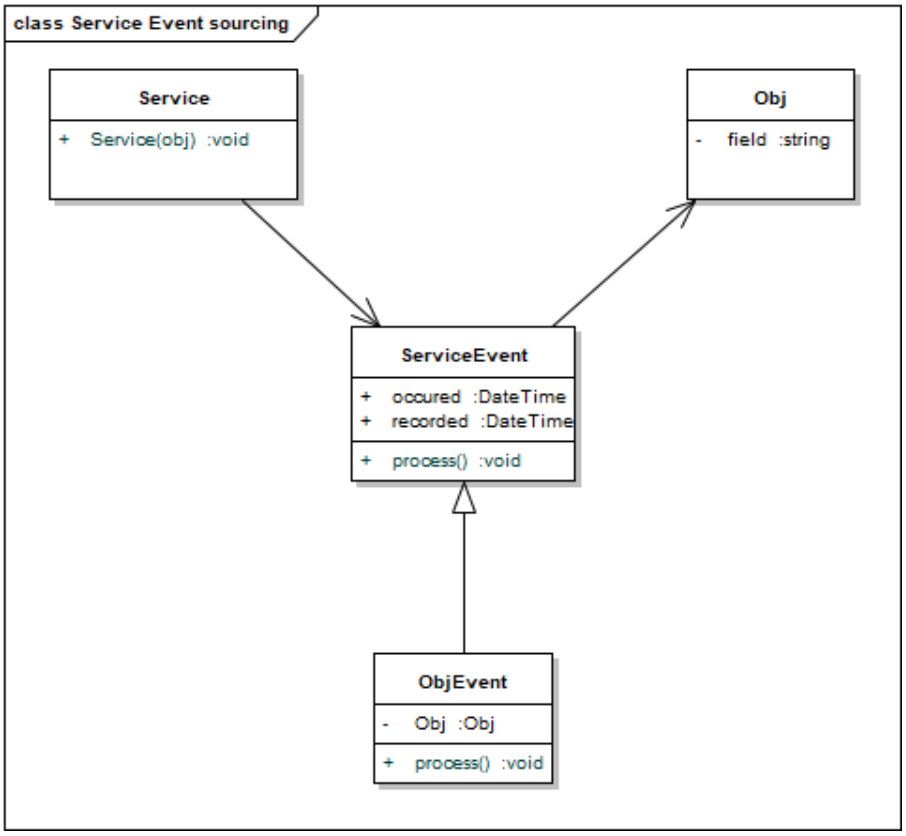
3.3 Struktur

Event Sourcing består af en sekvens af events, som er gemt i en Event Store af et eller anden format (SQL, noSQL, etc.), og bliver som regel brugt som en log, hvor i klasser kan se historiken af objekter, via. events.



Figur 3.1: Simpel service model

På figur 3.1, er der en simpel service, der gør et eller andet med et objekt, hvad den gør er underordnet, men den påvirker Obj på en eller anden måde. I et system uden Event Sourcing, vil det tidligere stadie af objektet være destrueret og glemt.



Figur 3.2: Simpel service med Event sourcing

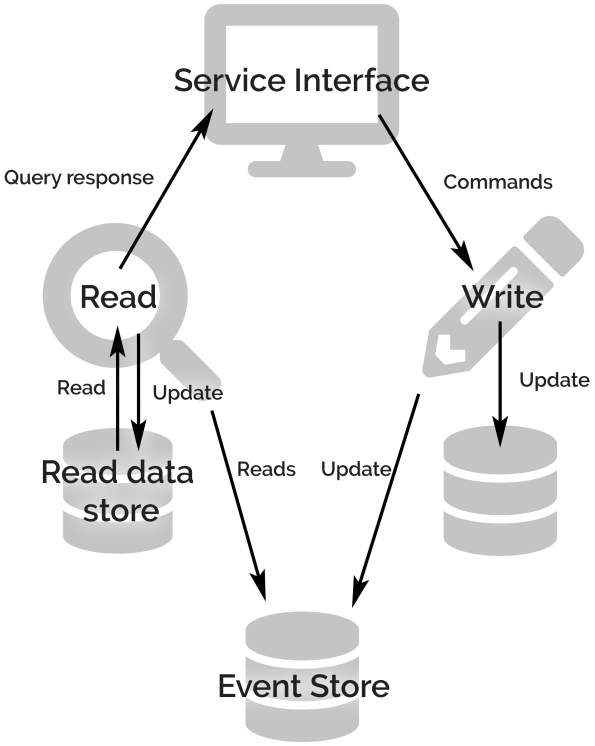
På figur 3.2 er der introduceret et event med et ServiceEvent, som kan bruges til at gemme stadiet af denne handling. Selve funktionaliten er ikke ændret, men Handlingen kan nu persistesteres.

3.4 Konsekvens

Ved implementering af Event Sourcing opnås et system, der kan reagere på hændelser, som dette lytter på. Dette kan sørge for, at ved at lytte på disse events kan opnå den ønskede funktionalitet, når f.eks en skriving til en database er foretaget.
Det kan om Event sourcing tilføjes, at disse events bliver rejst når en begivenhed indtræffer, men at det ikke nødvendigvis betyder, at der ikke samme tid er et nyt event på vej. Dette kan lede til inkonsistent data, hvor der læses noget data, der kan være ændret siden eventet der reageres på er indtruffet.
Dertil kan det være, at Event Store'en indeholder så mange events, at et sytem kan tage lang tid at initiere, da mange handlinger kan være foretaget. Dette kan dog løses ved at inddrage **Snapshots**, som sørger for, at sætte et state, hvor alle de nødvændige ting er til rådighed. Derefter kan de efterfølgende events udføres og dataen opsummeres.
Ved implementering af Event sourcing stiger kompleksiteten af systemet, som tidligere nævnt, hvilket betyder at Event Sourcing ikke altid er foretrukket.
Herunder er der listet nogle eksempler på systemer, hvor Eventsourcing bør overvejes.

- Applikationer, hvor der kan drages direkte paralleer til den event drevne virkelighed
- Applikationer, hvor det er en nødvendighed, at det ikke opstår konflikt ved dataopdatering
- Applikationer, hvor en opdatering, ved ny tilgængeligt data, er en nødvendighed
- Applikationer, hvor muligheden for at ramme et forgående stadie er en nødvendighed.
- Applikationer, hvor kompleksiteten i systemet og domænet i forvejen er høj.

Event Sourcing er ofte brugt sammen med **CQRS**, da dette kan medvirke til at dette pattern bliver mere indbringende, da skrive- og læseoperationerne bliver koblet i en Event Store, og derfor kan sørge for at disse har kendskab til hinandens stadier.



Figur 3.3: Simpel CQRS med Read og Write

[1]

På ovenstående figur 3.3 er det vist, hvor Event Sourcing tilføjes til **CQRS**-pattern'et, og hvordan dennes effekt indtræffer. Her ses det at skrivesiden kan opdatere Event Store og dermed notificere læsesiden, som nu ved at der er ny data tilgængelig, og kan nu foretage en ny iteration af Event Store'ens events.

[1] FiXme Note: Ændre til nyt billede hvor ES er tilføjet og ændre billedtekst

CQRS Implementering 4

Demonstration 5

6 Diskussion

Når man tilføjer **CQRS** og Event Sourcing til et komplekst software system, med mange objekter, data modeller, historik og samtidige problemstillinger, kan **CQRS** gøre overblikket nemmere, da man har mulighed for at dele de enkelte dele ud til flere udviklere, til gengæld bliver kompleksiteten for de individuelle elementer således større. Det kræver at man som udvikler tænker lidt anderledes i forhold til at være problemløsende, men baggrunden er at man kan uddele opgaver til andre udviklere, som ikke kender til den forretningsmodellen. Ved at bruge Event Sourcing, bliver systemet dog en del mere komplekst, da man med Event Sourcing laver versions historik, når man skriver. Event Sourcing vil derfor indføre et sværere system at implementere, da alle læs og skrive dele, stemples med versionshistorik. Her er GitHub et godt eksempel, da de har mange API til forskellige eksterne programmer, altså forskellige services. Alt bliver dokumenteret, dog kun på skrive siden, derfor kan man argumentere for at GitHub er Event Sourcing. Der er mange systemer som har fungeret på samme måde i rigtig mange år f.eks Bankverdenen, hvor man altid har kunnet se transaktioner. Hvilket også er grunden til at Event Sourcing og **CQRS** hænger så godt sammen.
Derudover er der flere fordele, ved at have flere services der læser eller bliver skrevet til:

- Ofte bliver der læst mere data end der bliver skrevet.
- Når vi læser data, får vi større mængder data ind. Når vi skriver, skriver vi altid kun til enkelte elementer af data, som kun påvirker ét aggregat.
- Fra en slutbrugers synspunkt, skal det at kunne læse data altid yde bedre end når vi skriver. Da en slutbruger typisk nemmere kan acceptere at vente på at skrive, end at vente på at kunne læse.

Ideen med at dele læsning og skriving af data op, bliver i større designløsninger mere relevant. I dag bliver mange systemer større og mere komplekse, argumentet for det er samme som virksomhederne, der bruger disse systemer. Enten fordi der er flere brugere eller fordi virksomheder går sammen for at spare midler og udnytter fælles synergier. Bliver store datamængder brugt i større systemer, ydeevne er foretrukket, vil det være en stor fordel at udnytte **CQRS**. Fordelen udviklingsmæssigt, er at man på den måde har mere overblik, da flere services der læser kan håndtere det samme data, og på samme måde, når der skal skrives. Hvilket også argumenterer for hvorfor **CQRS** for sig selv er relativt simpelt. Men når Event Sourcing implementeres vil de services der udfører operationerne, lave events så der på den måde er versionshistorik. Vi læser mere ved hjælp af API'er end vi nogenside har gjort, og i den forbindelse er **CQRS** en fordel. Komplekse systemer der har software enheder, der kun indeholder data, og hvor andre elementer er delt op, er i sidste ende muligheden for at lave service på en del, uden det nedlægger hele systemet, hvor overnævnte eksempel med API'er mellem systemer bliver mere og mere relevant.
Overordnet bruges **CQRS** ligeså meget et forretnings drevet design, som det er domæne drevet design. Da grunden til man vile implementere denne til fordel for **CRUD**, typisk er på baggrund af virksomhedes forretning, og det virksomheden kræver af software modellen.

7 Konklusion

Gennem arbejdet med **CQRS** er der opnået forståelse for, hvornår et pattern bliver til et Architectual Pattern og hvornår det forbliver et Design Pattern.
CQRS er i sin forholdsvise enkelthed et ret simpel Architectual Pattern, hvor man i nogle situationer bliver nødt til, at løfte dette op til et mere komplekst pattern ved at udvide implementeringen med event sourcing.

Eftersom **CQRS** ligger i spændingsfeltet mellen førnævte kompleksitetsgrader har det i særdeleshed været spændende at følge med i, hvornår denne linje krydses. Det har været meget indbringende at skulle definere, hvornår det forholdsvist simple **CQRS**-pattern har skulle udbygges med Event Sourcing som dels øger brugbarheden, ved nogle scenarier, men også hvordan denne udbygning kan have betydning for kompleksiteten.

Selve ideen om **CQRS** er ikke omfattende at forstå og implementere, men denne har også sine begrænsninger, som kommer i spil ved de uvisse stadier på dataen. Disse uvisheder kan afhjælpes ved, at implementere **CRUD** istedet for **CQRS**, hvor Event Sourcing tilføjes. Gennem udarbejdelsen af denne opgave er det blevet defineret hvor man bør gøre hvad og hvornår.

Der er ingen tvivl om at **CQRS** ligger inde med nogle fordele, som ikke kan benægtes. Blandt andet det faktum, at udviklingen af læse- og skriveoperationer kan foregå særskildt har stor betydning, da der er uligevægt i mængden af læse- og skriveoperationer. Denne særskilte implementering hjælper også på eventuelle skalerings problemer, der kan forekomme ved brug af **CRUD**.

Det har i særdeleshed været indbringende at prøve kræfter med, at implementere **CQRS**, især ved tilføjelsen af Event Sourcing. Implementeringen af **CQRS** som design pattern har medvirket til en forståelse af, hvor relativt simpel pattern'et kan være, men også hvor dette har sine udfordringer. Nogle af disse udfordringer har kunne afhjælpes ved, at lave implementeringen, hvori der også indgår Event Sourcing, men dette har givet anledning til, at skulle tænke på en andeledes måde, end man har været vant til. Eftersom denne Event drevne tankegang afviger fra den domænedrevene tankegang, som man ellers har brugt. Den eventdrevne tankegang har dog en kort tilvænningsperiode, da den ligger sig meget tæt op af virkelighedens events. På den måde kan der drages paralleller fra implementeringen, til den virkelige verden, hvilket har været en stor hjælp.

Bibliografi

Microsoft, Docs (2017). *CQRS*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs (sidst set 13.04.2018).

Wikipedia (2014). *CQS*. URL: https://en.wikipedia.org/wiki/Command%E2%80%93query_separation (sidst set 13.04.2018).

Rettelser