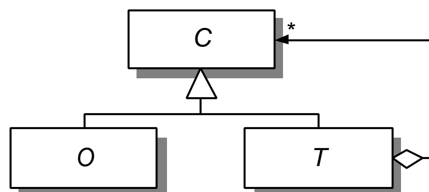
Mønstre

– en indføring i analyse-, design- og arkitekturmønstre

COT/4-07-V2.2



Center for Objekt Teknologi

Center for
Objekt Teknologi

Revisionshistorie:	12.01.99	v.0	Første udgave
	9.02.99	v.1.0	Anden udgave
	27.04.99	v.2	Endelig version
	10.05.99	v.2.1	Endelig version, med rettelser
	27.07.99	v.2.2	Udvidet version, med database-mønstre

Forfattere Aino Cornils, Aarhus Universitet
Johnny Olsson, WM-data
Ole Vedel Villumsen, WM-data

Status: Færdig

Offentliggørelse: Offentlig

Sammendrag:

Denne rapport giver en introduktion til begrebet mønstre.
Den indeholder en teoretisk gennemgang af begrebet, lidt historie og eksempler på praktisk anvendelse af mønstre. Vi har medtaget eksempler på tre typer mønstre; analysemønstre, designmønstre og arkitekturmønstre. Rapporten henvender sig til udviklere med kendskab til objektorienteret systemudvikling.

© Copyright 1999
Aino Cornils, Aarhus Universitet,
Johnny Olsson, WM-data og
Ole Vedel Villumsen, WM-data

Center for Objekt Teknologi (COT) er et projekt til forskning i, samt modning og indførelse af objektteknologi i danske virksomheder. Projektet er støttet af Center for IT-forskning og Erhvervsfremmestyrelsen.

Deltagere i projektet er:
Maersk Line, Maersk Training Center, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Stålskibsværft, A.P. Møller, Aarhus Universitet, Odense Universitet, Københavns Universitet, Teknologisk Institut og Dansk Maritimt Institut

Forord

Denne rapport er udarbejdet i et samarbejde under Center for Objekt Teknologi, COT, mellem Aarhus Universitet, WM-data og Teknologisk Institut.
Vi ønsker at takke Henrik Røn, Flemming Nielsen, Hanne Buhl, Jørn Heideman, Torben Bach Christensen, Kirsten Hjerrild Nielsen, René Elmstrøm, Egil Andersen, Jesper Buhl, Ellen Agerbo og Jesper Petersen for mange konstruktive kommentarer.
En særlig tak til Lisbeth Bergholt, Teknologisk Institut, for mange frugtbare diskussioner ved tilblivelsen af denne rapport.

1. INDLEDNING 5

1.1 OVERBLIK 5

2. EKSEMPEL PÅ BRUG AF ET MØNSTER 6

3. MOTIVATION 8

3.1 ERFARINGSFORMIDLING 9
3.2 FÆLLES ORDFORRÅD 9
3.3 DOKUMENTATION 9

4. MØNSTRE OG OBJEKTORIENTERET SYSTEMUDVIKLING 10

4.1 UDVIKLINGSPROCESSEN 11
4.2 ANALYSEMØNSTRE 13
4.3 DESIGNMØNSTRE 15
4.4 ARKITEKTURMØNSTRE 15
4.5 DATABASEMØNSTRE 16

5. ERFARINGER MED MØNSTRE 17

6. HVORDAN MAN KOMMER VIDERE 19

A. UML 20

B. CASE: UDVIKLING AF ET PLANLÆGNINGSSYSTEM 21

C. EKSEMPLER PÅ MØNSTRE 25

C.1 ANALYSEMØNSTRE — ET EKSEMPEL 25
C.2 DESIGNMØNSTRE — ET EKSEMPEL 27
C.3 ARKITEKTURMØNSTRE — TO EKSEMPLER 28
C.4 DATABASEMØNSTRE — 2 EKSEMPLER 30

D. LITTERATURLISTE 33

1. Indledning

Inden for softwareudvikling kan der spares tid ved genbrug, ikke bare af kode, men også af mere abstrakte ting som ideer. Erfarne udviklere genbruger egne systemudviklings-ideer idet de genkender problemer de har haft før, og bruger den løsning der, ifølge deres erfaring, virker bedst. Visse af disse ideer viser sig som tilbagevendende mønstre i gode designs. Derfor fik det begreb der blev introduceret i edb-verdenen, navnet „patterns", på dansk „mønstre". En karakteristik der ofte gives af mønstre, er at de er *abstrakte beskrivelser af løsninger på ofte tilbagevendende problemer*.

Vi vil her fokusere på brugen af mønstre ved at give en beskrivelse af hvad begrebet dækker over, og en forklaring af hvordan og hvornår mønstre kan anvendes i en udviklingsproces. Vores målgruppe er udviklere med edb-baggrund der ønsker at lære at anvende mønstre i deres arbejde. Kendskab til objektorienteret systemudvikling er en fordel fordi nogle mønstre er udviklet til objektorienteret brug. Vi vil senere vende tilbage til hvilke og hvorfor.

Indenfor mønstersamfundet opererer man med nogle flere begreber end vi har valgt at behandle i denne rapport (f.eks. framework og pattern language). Når vi går uden om disse, er det dels fordi der hersker stor begrebsforvirring omkring dem, dels for at vores fremstilling ikke skal blive for omfattende.

Vores mål er at forbedre udvikling af softwaresystemer ved hjælp af mønstre. Erfaring med brug af mønstre viser at brugen af mønstre letter og fremskynder arbejdet i en udviklingsproces og fører til et mere genbrugeligt design. Inspireret af dette har vi lavet denne appetitvækker til folk som ikke har prøvet at arbejde med mønstre før.

1.1 Overblik

Vi tager udgangspunkt i tre bøger og supplerer med to artikler.

Bøgerne er *Analysis Patterns* af Martin Fowler [Fowler 1997], *Design Patterns — Elements of reusable Object-Oriented Software* af Gamma, Helm, Johnson og Vlissides, populært kaldet GoF-bogen (fordi dens forfattere kaldes Gang of Four, firebanden) [Gamma 1994] og *Pattern-Oriented Software Architecture* af Buschmann med flere [Busch 1997].

Tilsammen indeholder bøgerne en mængde mønstre der dækker over både analyse- og designdelen af en udviklingsproces. Mønstrene er gennemarbejdede fra forfatternes side, og fordi bøgerne har været populære og anvendt i et par år, er mønstrene også

gennemprøvede af udviklere. Dette har afstedkommet en erfaringsopsamling som er beskrevet i afsnit 5.

De tre bøger indeholder forskellige typer af mønstre; [Fowler 1997] indeholder analysemønstre, [Gamma 1994] indeholder designmønstre, og [Busch 1997] indeholder såvel designmønstre som arkitekturmønstre. Analysemønstre fokuserer på begreber og sammenhænge fundet i problemområdet, designmønstre fokuserer på design af klasser/objekter og sammenhænge mellem disse, og arkitekturmønstre fokuserer på den overordnede struktur af designet, skelettet af systemet. Forskellen mellem de tre typer vil senere blive forklaret i større detalje. Vi vil give eksempler på de tre former for mønstre og brugen af dem og et eksempel på brug af alle tre slags i samme design.

De to artikler vi supplerer med, er *Crossing Chasms — A Pattern Language for Object-RDBMS Integration* af Kyle Brown og Bruce Whitenack [Chasms 1995] og *A Design Cookbook for Business Information Systems* af Wolfgang Keller [Keller 1996]. (Begge artikler ligger på world wide web, se litteraturlisten.) De fokuserer på integration mellem et program (som i det ene tilfælde er objektorienteret) og en relationel database, et emne som vi har ment at mange læsere ville interessere sig for. Mønstrene i de to artikler kalder vi under ét for databasemønstre.

Afsnit 2 giver et udførligt eksempel på brugen af et designmønster. Afsnit 3 beskriver fordelene ved at anvende mønstre. Afsnit 4 beskriver de tre typer mønstre, analyse-, design- og arkitektur-mønstre, og foreslår anvendelsen af dem i forskellige dele af en udviklingsproces. Derudover motiveres og beskrives databasemønstre (som vi ikke betragter som en yderligere type mønster). Afsnit 5 indeholder tilbagemeldinger fra designere der har arbejdet med mønstre. Afsnit 6 er en kort guide til hvordan man kommer videre med mønstre efter denne rapport. Appendiks B giver et stort eksempel hvori der indgår anvendelse af de førnævnte tre typer mønstre. Appendiks C indeholder eksempler på analyse-, design-, arkitektur- og databasemønstre.

Som notation i denne rapport er brugt UML, „Unified Modelling Language“ version 1.0 [Rum 1998]. UML-notationen er beskrevet i appendiks A. I de bøger og artikler vi har set på mønstre fra, benyttes forskellige notationer. Så når figurerne i rapporten ikke er nøjagtig som i bøgerne, er det fordi vi ved at „oversætte“ dem til UML undgår at beskrive og anvende forskellige notationer.

2. Eksempel på brug af et mønster

Forestil dig at man skal lave et tekstbehandlingssystem der kan lave dokumenter som indeholder både tekst og grafik. Systemet skal være effektivt, brugervenligt og med mulighed for senere udvidelse af funktionaliteten. Idet systemet skal være et

WYSIWYG[1]-system, skal brugeren løbende kunne se hvordan det færdige dokument kommer til at se ud, og kunne ændre i det på en simpel måde. F.eks. kunne han lave en figur i sit dokument ved at lægge et grafisk billede ind og få en kasse tegnet rundt om det og en tekst placeret under figuren. Når han har fået det på plads, vil han gerne kunne skifte størrelse på figuren som helhed, altså lade kassens størrelse følge figurens. Han vil også kunne flytte figuren uden at teksten stadig står på den plads han flyttede figuren fra. Han vil med andre ord behandle noget der består af flere enheder, som en helhed.

Af ovenstående følger denne kravspecifikation:

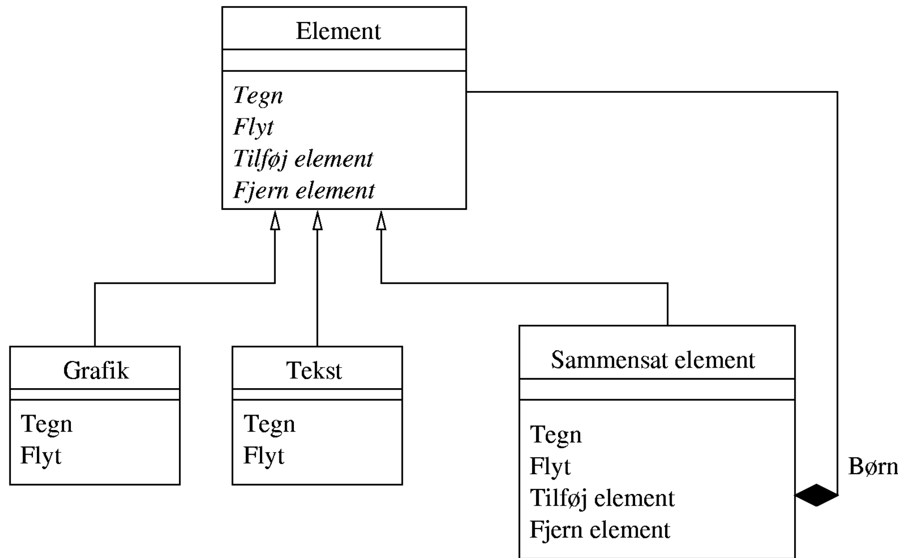
1. Tekst og grafik skal kunne bruges på samme måde og have samme prioritet; dvs. grafik er ikke en specialisering af tekst eller omvendt.
2. Sammensatte elementer og enkelte elementer skal behandles ens. De skal have det samme interface så der kan opereres med dem uden viden om hvilken slags element der arbejdes på.
3. Forskellige typer af enkelte elementer skal kunne analyseres på forskellig måde. F.eks. skal stavekontrol kunne udføres på en tekst, men ikke på en cirkel.

En måde at leve op til kravspecifikationen er at lave en rekursiv sammensætning af de enkelte komponenter i dokumentet. Som et første skridt kan vi sætte bogstaver og grafik sammen fra venstre til højre for at danne en linje i dokumentet, flere linjer kan grupperes sammen til søjler, og flere søjler kan samles til en side i dokumentet o.s.v. Denne fysiske struktur kan repræsenteres ved at knytte et objekt til hvert element i dokumentet. Ikke bare de synlige elementer, bogstaverne og grafikken, men også søjlerne, linjerne og siderne.

Klassediagrammet for dette design kan ses af figur 1. De enkelte elementer, såsom tekst og grafik, har deres egne klasser mens de sammensatte elementer, såsom linjer, søjler og sammensatte figurer, har en fælles klasse, *sammensat element.* Alle disse klasser er specialiseringer af klassen *element*, hvilket giver et fælles interface til alle slags elementer. Sammensatte elementer har tre basale ansvarsområder. De ved hvordan de skal tegne sig selv, de ved hvor de skal tegnes, og de kender de enkelte elementer de er bygget op af, børnene. Når et sammensat element får besked af den del af systemet der tager sig af brugerønsker, på at tegne sig selv eller ændre sig, vil det sende beskeden videre til alle de elementer det er bygget op af, så de også bliver tegnet/ændret som påkrævet.

Med denne struktur lever systemet op til brugerkravene. Tekst og grafik ligger på samme niveau i klassehierarkiet, der er samme interface til enkeltstående og sammensatte elementer, og det er nemt at udvide systemet med flere typer elementer.

[1] What You See Is What You Get



Figur 1. Klassediagrammet for designet til teksteditoren. Sammensatte elementer som linjer, søjler og sider består af samlinger af enkelte elementer som grafik og tekst. De har selv ansvaret for at tilføje og slette de elementer de er opbygget af.

Denne løsning på et designproblem findes i designmønsteret „Composite“ som er beskrevet i GoF-bogen. En beskrivelse af et GoF-mønster består af flere dele. Den løsning der ligger i mønsteret, og de fordele og ulemper der er ved at anvende det, er mestendels beskrevet tekstuelt. Fordelene ved brug af Composite-mønsteret har vi været inde på. Som et eksempel på en ulempe ved mønsteret kan nævnes at det er svært på kørselstidspunktet at begrænse et sammensat element til kun at kunne sammensættes af bestemte typer delelementer da det er lavet så generelt at alt kan sættes sammen. Udover en tekstuel beskrivelse gives en grafisk beskrivelse af strukturen som den der er vist i figur 1 der kan give en hurtig, men overfladisk forståelse af løsningen. Bogen giver også implementationsideer og henvisninger til andre mønstre der er relaterede til det. Desuden gives henvisninger til eksempler på brug af mønsteret. Da mønstre skal have vist sig som ofte forekommende løsninger, mener forfatterne af GoF-bogen at der skal kunne henvises til mere end én anvendelse af mønsteret før der er tale om et mønster.

3. Motivation

Det at bruge mønstre til at beskrive abstrakte løsninger på tilbagevendende problemer inden for objektorienteret design, har vist sig at have følgende fordele:

- Erfaringsformidling
- Fælles ordforråd

- Bedre dokumentation

Naturligvis er der også ulemper forbundet med anvendelsen af mønstre. Den største ulempe er at udviklere der lige er begyndt at anvende mønstre, har en tendens til „overforbrug“ af mønstre. Man bliver så imponeret af ideen at man ønsker at bruge mønstre til alle designbeslutninger. Men der findes ikke en mønsterløsning til alle problemer. Og det er ikke meningen at man skal få sit problem til at passe til et mønster bare for at kunne bruge mønstre, men at man kan bruge et mønster fordi det passer til ens problem. I beskrivelserne af mønstrene findes også forslag til variationer af mønstrene.

3.1 Erfaringsformidling

Ligesom i andre discipliner indenfor systemudvikling findes der problemer der forekommer oftere end andre indenfor objektorienteret design. Mange af disse problemer har en umiddelbar løsning der kan have uheldige konsekvenser. Det er svært at forudse konsekvenserne af en løsning på forhånd, og uden erfaring i at løse designproblemer kan ens løsninger tit ende med at blive de umiddelbare, dårligere løsninger. Når en systemudvikler har opnået en vis erfaring, vil han kunne genkende nogle mønstre i problemer og løsninger og umiddelbart kunne anvende den bedste løsning han kender. Hvis han beskriver denne løsning som et mønster, vil andre udviklere kunne bruge hans erfaring og spare tid fordi de ikke behøver genopfinde de gode løsninger.

3.2 Fælles ordforråd

Udover at erfaring bliver nedskrevet og formidlet på en præcis måde, vil udviklerne ved hjælp af mønstre kunne diskutere udvikling på et mere abstrakt niveau end normalt fordi et mønster dækker over et helt lille deldesign og mønstrene kan identificeres ved hjælp af deres navn. Når systemudviklere der kender til mønstre, diskuterer forskellige potentielle løsninger på et problem, bruger de navnene på mønstrene til at kunne kommunikere om komplekse ideer på en præcis og kortfattet måde. Hvis mønstre bliver alment kendte og brugt i både undervisningen af udviklere og det daglige arbejde, vil disse mønstre kunne bruges i samme grad som algoritmer og datastrukturer bruges indenfor andre programmeringssprog.

3.3 Dokumentation

Med brug af mønstre kan dokumentation af softwaresystemer effektiviseres da det er muligt at udtrykke sig præcist med få ord. Det er nemmere og hurtigere at angive anvendelsen af et bestemt design ved at skrive et navn på et mønster i dokumentationen end ved at beskrive alle dele af, og relationer i, designet. Det er nok at skrive hvilken

rolle en bestemt klasse eller et bestemt delsystem spiller i et mønster. Denne kortfattede information vil give læseren oplysninger om hvilke andre klasser denne klasse arbejder sammen med, og hvilket ansvar, eventuelt hvilket interface, den har.

En anden fordel er at vedligeholdelsesbyrden reduceres ved at bruge mønstre. Mønstrene stiler mod fleksibelt design og gør det dermed nemmere at vedligeholde systemerne. Når mønstre anvendes i en udviklingsproces, vil den overordnede idé løbende kunne beskrives ved hjælp af mønstre. Den derved nedskrevne viden ville normalt have været implicit hos udviklerne, men det at gøre den eksplicit gør det lettere at bruge den. Udviklerne følger dermed den samme overordnede designidé, og dette gør designprocessen mere effektiv.

For ethvert mønster er der en beskrivelse af sammenhængen; en beskrivelse af hvor dette mønster med fordel kan anvendes. Ved at skrive at et bestemt mønster er valgt, skrives dermed også, i hvert fald i grove træk, hvilket problem der førte til at mønsteret blev valgt. I traditionel dokumentation findes som oftest ikke nogen beskrivelse af hvad der førte til valget af en bestemt løsning. Hvis den der læser dokumentationen også kender til mønstre, vil vedkommende være i stand til at fange den indforståede information der ligger i at et bestemt mønster er valgt. For at kunne få fordel af dokumentation med mønstre kræves det naturligvis at læseren af dokumentationen kender, eller i det mindste har adgang til, en beskrivelse af de mønstre der er anvendt.

Dokumentation ved brug af mønstre kan dermed give mere information end traditionel dokumentation, hvilket, uden ekstra arbejde, vil kunne forbedre dokumentationen. UML [Rum 1999] indeholder beskrivelser af anvendelser af designmønstre som umiddelbart kan bruges i dokumentationen. Dette er illustreret i appendiks A.

4. Mønstre og objektorienteret systemudvikling

For en snes år siden identificerede en arkitekt, Christopher Alexander [Alex 77], en samling mønstre der gik igen i arkitekturens verden. Han mente at disse mønstre beskrev endegyldige sandheder for hvad der er godt for mennesker at bo og leve i. Hans mål var at folk selv, ved hjælp af disse mønstre, skulle kunne designe deres huse og haver.

Omkring ti år efter introducerede et par dataloger denne idé for udviklere der brugte objektteknologi, ved at nedskrive løsninger på designproblemer på en homogen måde i artiklen: „Using Pattern Languages for Object-Oriented Programs“ [oopsla87]. I arkitekturens verden beskrives alt i termer af generelle begreber[2] og specifikke fænomener[3] ved hjælp af tekst *og* figurer. Denne beskrivelsesform er direkte oversættelig til

[2] Døre, vinduer og huse.
[3] Hoveddør*en*, Downing Street nr. 10.

objektteknologi, som behandler klasser og objekter. De første mønstre behandlede kun designproblemer på objekt/klasse-niveau, men da disse viste sig at være en succes, fulgte andre typer mønstre som analysemønstre og arkitekturmønstre.

Fælles for disse mønstre er at de har deres oprindelse indenfor objektorientering. Mønstrene er ikke udviklet ved at nogen har sat sig ned og tænkt sig frem til dem; de er blevet opdaget som mønstre der går igen i gode design- og analyse-modeller. De er identificerede i objektorienteret sammenhæng og er derfor tydeligt objektorienterede i deres løsningsbeskrivelse idet de i høj grad beskæftiger sig med objekters ansvarsområder og indbyrdes kommunikation. Det er derfor fristende at konkludere at mønstre kun kan anvendes indenfor objektorienteret teknologi, men dette er ikke nødvendigvis sandt.

Umiddelbart kan siges at analyse- og arkitekturmønstre, idet de bruger objekt-tankegangen på et meget abstrakt niveau, kan bruges indenfor alle sprogparadigmer.

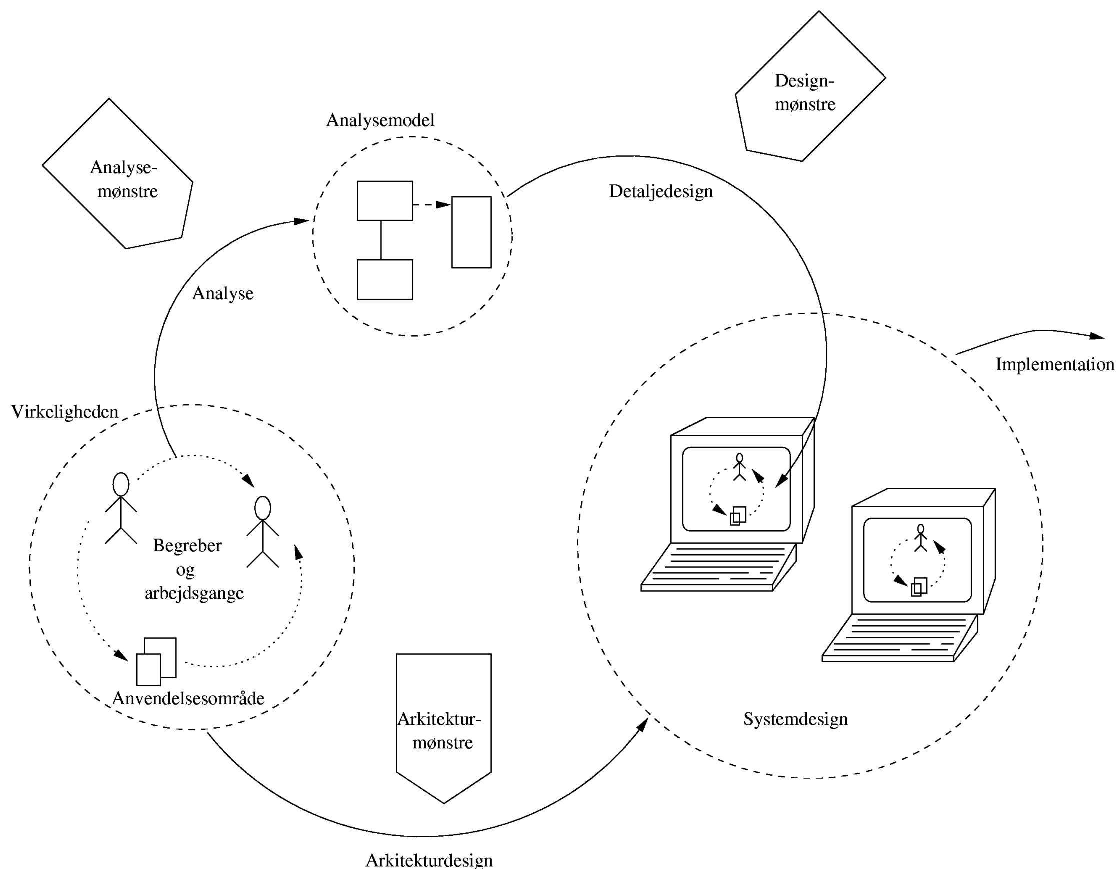
Hvad designmønstre angår, stiller sagen sig lidt anderledes. På den ene side er det rigtigt at mange designmønstre i deres løsningsbeskrivelse gør brug af objektorienterede sprogkonstruktioner som polymorfi og nedarving. På den anden side er disse ikke essentielle for implementationen af mønstrene. Nogle designmønstre kan fuldt ud implementeres i andre former for sprog mens andre bare mister en del af deres elegance ved det. Det kan blive nødvendigt at gå på kompromis med sine udtryksmuligheder afhængigt af hvilket værktøj der benyttes, men det er stadig muligt at bruge de gode ideer der ligger i dem, indenfor andre sprogparadigmer såsom funktionelle eller imperative sprog.

4.1 Udviklingsprocessen

Figur 2 viser de forskellige delaktiviteter af en udviklingsproces og hvor de forskellige typer mønstre kan bruges i udviklingsprocessen.

Opdelingen i delaktiviteter er ikke gjort ud fra en overbevisning om at systemudvikling består af isolerede, selvstændige arbejdsprocesser, men udfra et fremstillingsmæssigt synspunkt. For at vise at forskellige typer mønstre er egnede til forskelligt fokus i udviklingsprocessen, er det nødvendigt at have en distinktion mellem de delprocesser der udgør udviklingsprocessen.

I figuren er tre delmål repræsenteret ved tre cirkler. En af dem symboliserer den afgrænsning af virkeligheden man forsøger at udvikle et system til. I denne indgår både de begreber og arbejdsgange der findes i problemdomænet, og det anvendelsesområde systemet er tiltænkt, altså ydre betingelser. En anden cirkel viser analysemodellens rolle i udviklingsprocessen. En analysemodel er en model der beskriver de objekter og arbejdsgange der er nødvendige for systemet, identificerede i problemdomænet. Den sidste cirkel repræsenterer det endelige systemdesign. Dette er baseret på problemdomænet og analysemodellen. Ud fra dette design kan systemet umiddelbart implementeres.



Figur 2 Delprocesserne i en udviklingsproces med udgangspunkt i anvendelse af mønstre. Pilene repræsenterer delprocesser i systemudvikling mens cirklerne repræsenterer henholdsvis den del af virkeligheden der er interessant for systemet, og de to delresultater, analysemodellen og systemdesignet.

Pilene viser de delaktiviteter hvor mønstre hjælper udviklingen. Det er vigtigt at huske at det kan være nødvendigt at genanalysere sit problemdomæne i designfasen, og at arkitekturen af ens system ikke nødvendigvis er bygget efter de arbejdsgange eller ansvarsfordelinger der findes i problemdomænet. Så når en pil peger fra analysemodellen til systemet er den ikke i praktisk henseende retningsangivende for udviklingen, men den mest illustrative præsentation.

Arkitekturdesign-pilen går fra problemdomænet til systemdesignet for at vise at arkitekturmønstre kan anvendes til at designe den overordnede arkitektur af systemet udfra problemdomænet. Det analyseres på et overordnet niveau hvor systemet inddeles i delsystemer og deres ansvarsområder defineres. Denne overordnede struktur udgør skelettet i systemdesignet. Analysepilen går også fra problemdomænet, men her er analysen på et mere detaljeret niveau hvor de enkelte objekter bliver identificeret og får tildelt ansvarsområder.

Brugen af analysemønstre på dette niveau giver en analysemodel som resultat. Denne model er en abstrakt og formel beskrivelse af analysen af problemdomænet, og udfra denne kan systemet beskrives ved en designmodel der umiddelbart kan implementeres, som illustreret ved detaljedesignpilen. Den sidste pil på figuren, implementationspilen, indikerer at selve implementationen af systemet er udenfor mønstre fra denne rapports fokus. Hvordan et systemdesign implementeres afhænger i høj grad af valg af implementationssprog.

De følgende fire afsnit, 4.2–4.5, er skrevet med basis i bøgerne [Fowler 1997], [Gamma 1994] og [Busch 1997]'s formidlingsmetode samt i de to artikler [Chasms 1995] og [Keller 1996]. De beskriver kategorierne af analyse-, design- og arkitekturmønstre som de er beskrevet i bøgerne og databasemønstre som de er beskrevet i de to artikler. Eksempler på mønstre kan findes i appendiks C.

I afsnit 4.2–4.4 bruger vi den skarpe opdeling af de tre typer mønstre for at kunne beskrive hvilket fokus de har på udviklingsprocessen. Visse analyse- og designmønstre kan minde meget om hinanden, men deres løsninger henvender sig til to forskellige abstraktionsniveauer som vist i figur 2. Design- og arkitekturmønstre kan også være svære at skelne fra hinanden. Begge typer løsninger ligger indenfor det endelige systemdesign. Den eneste tydelige forskel ligger i at en ændring i valg af arkitekturmønster efter implementation har større omkostninger end en ændring i valg af designmønster. Dette skyldes forskellen i abstraktionsniveau af de indgående elementer i de to typer mønstre. I et arkitekturmønster er det hele delsystemer der arbejdes med, mens det i designmønstre er klasser og objekter.

Efter de tre afsnit om analyse-, design- og arkitekturmønstre følger afsnit 4.5 om databasemønstre. Databasemønstre er en betegnelse for visse design- og arkitekturmønstre som finder anvendelse i forbindelse med tilgang til relationelle databaser. Her fraviger vi altså opdelingen i de tre typer mønstre og samler beskrivelsen af disse i et afsnit for sig selv.

4.2 Analysemønstre

Analysemønstre som de beskrives i Fowlers bog [Fowler 1997], afspejler begrebsmæssige strukturer i problemdomænet mere end de beskriver aktuelle softwareimplementationer. Et analysemønster beskriver et typisk problem, en del af virkeligheden som umiddelbart kunne være blevet modelleret på en besværlig og ineffektiv måde. Dernæst giver det en løsning på hvordan en mulig model kunne se ud, og beskriver de problemer der måtte ligge i denne model. Det at bruge mønstre i analyse er en anden måde at gribe tingene an på end man hvis man bruger de teknikker til analyse der blev brugt allerede før mønstre blev introduceret. Analysemønstre giver hele løsninger mens ældre analysemetoder består af værktøj til at udlede løsningerne af problemområdet. Dette ses i Rumbaughs bog [Rum 1991], der beskriver teknikker og notation til analyse,

og i Matthiasens bog [Mat 1997], der beskriver analyseaktiviteter. Et eksempel på et analysemønster findes i appendiks C.1.

Fowler mener at analysemodeller er vigtige for udviklingsprocessen fordi software-udvikling bør afspejle menneskelig tænkning, og ikke omvendt. Det er systemerne der skal laves så de passer til os, ikke os der skal tænke som systemerne.

Mønstrene i bogen deles i to; en del indeholdende analysemønstre og en del indeholdende såkaldte understøttelsesmønstre. Analysemønstrene beskriver begrebsmæssige mønstre fra forretningsmodeller. De kaldes begrebsmæssige fordi de beskriver hvordan folk tænker, snarere end hvordan et softwaresystem er designet. Understøttelsesmønstrenes formål er at vise hvordan analysemønstre kan anvendes, hvordan de passer ind i en systemarkitektur, og hvordan de begrebsmæssige modeller kan implementeres.
Vi vil her fokusere på selve analysemønstrene.

Mønstrene deles op i 9 kategorier, hver med fokus på sit problemdomæne. Analysemønstrene inddeles i disse domæner fordi det var der de blev identificeret, ikke for at afgrænse deres anvendelsesområde. Så læseren bør kigge efter analysemønstre indenfor alle domæner. De 9 domæner er:

1. Accountability: Organisering af strukturer for situationer hvor personer eller organisationer er ansvarlige for eller afhængige af hinanden.
2. Observations and Measurement: En model for repræsentation af observationer og målinger.
3. Observations for Corporate Finance: En udvidelse af mønstrene i førnævnte gruppe som gør at de kan anvendes indenfor finansanalyse.
4. Referring to Objects: En gruppe af mønstre der giver modeller til repræsentation af referencer til objekter.
5. Inventory and Accounting: En modellering af hvordan økonomien i et firma bevæger sig.
6. Planning: Planlægningsmønstre. Individuelle planer og mere generelle protokoller for planer.
7. Trading: Mønstre for modellering af køb og salg af varer.
8. Derivative Contracts: Modeller for handel hvor en pris afhænger af en anden.
9. Trading Packages: Hvordan en stor domænemodel opdeles i mindre dele.

Indenfor hvert domæne findes flere beskrivelser af mønstre hvor det første mønster der beskrives, er det simpleste. Martin Fowler bruger ideen i dette mønster til at bygge mere og mere komplicerede mønstre. I denne proces specialiserer han både problembeskrivelsen og løsningen. Dette giver en simpel indlæringsproces for læseren samtidig med at han bliver præsenteret for de mere komplicerede mønstre.

4.3 Designmønstre

Et eksempel på et designmønster findes i appendiks C.2. I indledningen til GoF-bogen [Gamma 1994] siger forfatterne at bogen, på trods af sin størrelse, kun indeholder en lille del af de mønstre en designer har brug for. Den indeholder f.eks. ingen mønstre der har med parallelitet, distribution eller realtidsprogrammering at gøre. Men de mønstre bogen *har* med, dækker over design-erfaring som kan være nyttig at have ved hånden ved design af gode, genbrugelige objektorienterede systemer.

Mønstrene i denne bog er alle på nogenlunde samme abstraktionsniveau. De er ikke opskrifter på linkede lister eller hashtabeller, der kunne være i en klasse og ligge i et bibliotek; de er heller ikke komplekse eller domænespecifikke designs for hele systemer. Designmønstre er beskrivelser af kommunikerende objekter og klasser der løser generelle designproblemer. De identificerer de deltagende klasser og objekter og deres roller, ansvarsområder og samarbejde.

Bogen er skrevet af folk med en baggrund i programmeringssprogene C++ og Smalltalk, og de løsninger de kommer frem til er dermed de løsninger der er nemmest eller overhovedet mulige i disse sprog. Dette er vigtigt at holde sig for øje hvis bogens mønstre bruges i et andet programmeringssprog.

Mønstrene er opdelt i tre kategorier; „creational“, *de skabende*, „structural“, *de strukturelle* og „behavioral“, *de opførselsmæssige*. De skabende mønstre tager sig af processen med at skabe objekter. De strukturelle mønstre beskæftiger sig med komposition af klasser eller objekter. De opførselsmæssige karakteriserer måderne hvorpå klasser eller objekter kommunikerer og samarbejder. *Composite*-mønsteret, som blev brugt i eksemplet i afsnit 3, er et strukturelt mønster fordi intentionen med det er at beskrive hvordan objekterne skal sammensættes. I appendiks C er et eksempel på et opførselsmæssigt mønster, *Mediator*. Dette beskriver hvordan objekterne i et design kan kommunikere.

Mønstrene i GoF-bogen inddeles ikke, som analysemønstrene, efter hvilke domæner de er identificeret i, men derimod efter hvilken type problemer de kan løse. Dette gøres, fordi forfatterne har forsøgt at vise mønstrene i en så neutral sammenhæng som muligt.

4.4 Arkitekturmønstre

I bogen „Pattern-oriented Software Architecture“ [Busch 1997] findes ikke bare arkitekturmønstre, men også designmønstre og implementationsmønstre (også kaldet idiomer). Vi vil kun behandle arkitekturmønstrene her. Eksempler på arkitekturmønstre findes i appendiks C.3.

Buschmann motiverer brug af arkitekturmønstre med at sige at levedygtige softwarearkitekturer er bygget ifølge nogle overordnede struktureringsprincipper. Disse principper kan beskrives af arkitekturmønstre. Et *arkitekturmønster* udtrykker et fundamentalt strukturelt organiseringsskema for softwaresystemer. Det giver et sæt prædefinerede delsystemer, specificerer deres ansvarsområder og inkluderer regler og retningslinjer for organiseringen af samspillet mellem dem. Arkitekturmønstre er skabeloner for konkrete softwarearkitekturer. De specificerer de overordnede strukturelle egenskaber af et softwaresystem og har en indvirkning på arkitekturen af dets delsystemer. Valget af arkitekturmønster er derfor en fundamental beslutning ved udvikling af et softwaresystem.

Arkitekturmønstrene i bogen er inddelt i 4 grupper efter hvilken type overordnet plan mønsteret følger. Den første, „fra mudderhul til struktur", fokuserer på at strukturere ellers forvirrende systemarkitekturer. Specielt understøtter mønstrene i denne gruppe en kontrolleret opdeling af det overordnede system i samarbejdende delopgaver. Den anden, „distribuerede systemer", indeholder mønstre til design af distribuerede systemer. Den tredje, „interaktive systemer", er til design af systemer med brugerinteraktion. Og den fjerde, „systemer der tilpasses", består af mønstre til design af fleksible systemer der skal kunne udvides og ændres nemt og effektivt.

I appendiks C har vi *Layers*, et mønster fra gruppen „fra mudderhul til struktur", og *Broker*, et mønster fra gruppen „distribuerede systemer". I appendiks B, hvor et eksempel på brug af alle tre typer mønstre er vist, bruges *Model-View-Controller* fra gruppen „interaktive systemer".

4.5 Databasemønstre

Mange systemudviklere har gennem tiden erfaret at det ikke er trivielt at tilgå relationelle databaser på en pæn og vedligeholdelsesvenlig måde, heller ikke fra objektorienterede programmer. F.eks. har nogle udviklere fundet det nødvendigt at forurene deres applikationskode med SQL-kode der er afhængig af detaljer i databasedesignet der burde være applikationens kerne uvedkommende; eller de har dubleret triviel databasetilgangskode næsten i det uendelige for at kunne tilgå mange tabeller i en database. Der findes en del mønstre der løser problemer i denne forbindelse. Sådanne mønstre har vi valgt at kalde *databasemønstre*. Databasemønstre er således ikke en fjerde slags mønstre ved siden af analyse-, design- og arkitekturmønstre. Tværtimod er ethvert mønster der bliver præsenteret i det følgende, enten et design- eller et arkitekturmønster. Men da disse mønstre hører sammen og bruges sammen, har vi valgt at præsentere dem i et afsnit for sig selv. Her vil vi kort se på mønstre fra de to artikler [Chasms 1995] og [Keller 1996]. Eksempler fra begge artikler findes i appendiks C.4.

Crossing Chasms [Chasms 1995] betyder at krydse kløfter; i dette tilfælde kløfterne mellem objektorienterede programmer og relationelle databaser. Mønstrene i artiklen er inddelt i tre grupper à ni mønstre:

1. De statiske mønstre: Syv af disse beskriver hvordan man repræsenterer en objektorienteret model i tabeller i en relationel database. Et par af de problemer der løses, er at objekter ikke har primærnøgler, men til gengæld har identiteter på en anden måde end rækker i en tabel. Der er f.eks. mønstre der beskriver hvordan man repræsenterer objekter, objektrelationer, specialiseringshierarkier og „collections" i tabeller. De sidste to statiske mønstre vedrører det objektorienterede program; det ene anbefaler f.eks. at man i programmet så vidt muligt bruger direkte referencer mellem objekter fremfor (databasens) fremmednøgler.
2. De dynamiske mønstre: en række mønstre til brug i et program i forbindelse med tilgang til databasen, herunder håndtering af hentning, gemning og transaktioner. Her håndteres bl.a. det problem at databasens relationelle model af verden ser anderledes ud end programmets objektorienterede model af den samme verden.
3. Client-server-mønstrene: Disse mønstre beskriver samspillet mellem et applikationsprogram og en database, f.eks. fordeling af opførsel (kode), synkronisering mellem data de to steder, låsningsstrategi og caching. Emnerne er relevante uanset om applikation og database kører på samme maskine eller (som ofte) fordelt på to eller tre maskiner.

I en senere artikel [Chasms 1998] har Kyle Brown præsenteret fire arkitekturmønstre til *Crossing Chasms*. Her beskæftiger han sig bl.a. med en moderne trelags- (three tier) arkitektur.

Artiklen [Keller 1996] er især beregnet på situationer hvor man gerne vil ændre databasedesignet af hensyn til tilgangstid efter at man er begyndt at udvikle applikationsprogrammer der bruger databasen. Under visse forudsætninger kan med anvendelse af mønstre fra artiklen holde database-tabel- og -feltnavn helt ude af koden til applikationsprogrammers kerne. Artiklen lægger desuden vægt på at de anbefalede løsninger kan implementeres i ikke-objektorienterede sprog. Artiklen præsenterer en ramme (framework pattern) kaldet *Relational Database Access Layer,* nogle strukturelle mønstre hvoraf de hierarkiske, logiske views og de fysiske views er centrale, samt et antal gængse databaseoptimeringstricks.

5. Erfaringer med mønstre

Dette og næste afsnit er skrevet med fokus på erfaring med og fordele af anvendelse af designmønstre. Der er tilsvarende erfaringsopsamlinger for analyse- og arkitekturmønstre, men vi har fokuseret på designmønstre her da det er den ældste og mest velbeskrevne form for mønstre indenfor systemudvikling.

Der findes en del forskergrupper som beskæftiger sig med mønstre. Den gruppe der blev stiftet først, kalder sig „The Pattern Community“. En del af forskerne i denne gruppe har samlet deres erfaringer med anvendelse af designmønstre i artiklen „Industrial Experience with Design Patterns“ [ICSE1996], som er tilgængelig fra mønstrenes hjemmeside [uiuc1999].

En af forskerne hvis arbejde er med i artiklen, er John Vlissides, som er en af forfatterne til GoF-bogen. I artiklen beskriver han hvordan det at begynde at anvende mønstre påvirkede hans arbejdsproces. Han havde arbejdet som konsulent for et halvt dusin firmaer i omkring 4 år da han i 1993 begyndte at anvende de rå mønsterbeskrivelser, herefter kaldet præmønstre[4], der senere skulle komme til at være en del af GoF-bogen, i sit arbejde. Der var to ting der havde irriteret ham gennem årene, og det var dem der fik ham til at begynde at anvende sine præmønstre.

For det første var det næsten umuligt for John Vlissides at få systemudviklerne til at beskrive deres design for ham. Både fordi de ikke havde en god måde at kommunikere designet på, og fordi de ikke var bevidste om relationerne mellem delsystemerne i designet.

For det andet havde udviklerne designet systemet uden at være opmærksomme på hvorfor de havde valgt et bestemt design. Årsagerne til de forskellige designbeslutninger var aldrig blevet husket endsige nedskrevet. Dette resulterede i at ændringer i kravspecifikationen ikke umiddelbart kunne overføres til designet fordi det ikke var oplagt hvor designet skulle ændres for at tilpasse det til ændringerne.

Da John Vlissides begyndte at anvende præmønstre, gjorde han det ved at sammenligne klienternes design med de præmønstre han kendte for design. Hvis deres design stemte overens med et præmønster han kendte, kunne han associere delene af deres design med delene i præmønsteret og se om de passede sammen. På den måde skabte han en referenceramme når de diskuterede design. Det blev lettere at kommunikere design, og det blev lettere at stille udviklerne de rigtige spørgsmål.

Artiklen indeholder udover seks indlæg om erfaringer en opsummering af fælles erfaringer. De vigtigste erfaringer der uddrages, er følgende: Mønstre er et godt kommunikationsmedie til gruppekommunikation fordi det giver udviklerne i gruppen fælles referenceramme med abstrakte, korte og præcise designbeskrivelser. Beskrivelse af design ved hjælp af mønstre gør det nemmere for udviklerne hele tiden at holde sig deres designbeslutning for øje. Denne viden er normalt implicit, og det at gøre den eksplicit, og dermed synlig, gør udviklingsprocessen mere effektiv. Mønstrene er velegnede til undervisning i softwaredesign. Erfaring og gode designideer kan beskrives på en præcis og letforståelig måde som gør det langt nemmere for udviklere at genbruge andre udvikleres erfaring.

[4] Ordet „præmønster“ er vores egen konstruktion.

I Danmark har flere arbejdspladser allerede forsøgt sig med brug af mønstre, og med stor succes. De erfaringer vi har hørt fra folk der har brugt dem, er følgende:
Hvis man som udvikler møder et mønster der indeholder en løsning man selv har brugt, vil man blive bekræftet både i at man i sin tid valgte en god løsning, og i at mønstre beskriver gode løsninger på tilbagevendende problemer. Man vil blive klar over at man da man fandt på løsningen, kunne have sparet den tid det tog en at komme på løsningen, hvis man havde kendt til mønsteret. Og i fremtiden vil man kunne spare tid ved at bruge mønsterets navn når man skal forklare andre udviklere hvad man har gjort, i stedet for at skulle forklare alle indgående roller og ansvarsområder.

6. Hvordan man kommer videre

Når anvendelsen af mønstre skal til at tage sin begyndelse, er det en god idé først at bruge lidt tid på at sætte sig grundigt ind i de mønstre man har valgt at bruge, og ikke bare slå op i en tilfældig bog om mønstre og bruge dem hovedløst. Det kan desuden være en god idé at kontakte en person der har forstand på mønstre; en mønster-mentor. En mønster-mentor kan bruges til at opmuntre udviklerne til at bruge mønstre og på samme tid sørge for at de bruger dem rigtigt. Det er et almindeligt begynderproblem at man bliver så begejstret for mønstre at man entusiastisk bruger dem til alting. Men selvom mønstre er gode, er de ikke gode i alle situationer. Det er ikke en god idé at prøve at ændre sine systemkrav så de kommer til at passe til en bestemt mønsterløsning.

Hvis det ikke er muligt at få en mønster-mentor, kan udviklerne lære at anvende mønstrene ved at læse beskrivelsen af dem og siden diskutere dem. Diskussionen kan føre til at udviklere indenfor arbejdspladsen får udbytte af hinandens problemer, løsninger og konsekvenser af løsninger.

Det er naturligvis ikke meningen at man skal kunne huske alle mønstrene udenad. Når man har fået en fornemmelse for hvad de hver især dækker over af løsninger, er det en god idé at benytte litteraturen som opslagsbog når man overvejer at bruge et mønster.

En god indgangsvinkel er at begynde at arbejde med mønstre i det små, altså med en overskuelig mængde i starten og langsomt udvide sit ordforråd med flere mønstre efterhånden som man arbejder med dem. Vi kan anbefale at læse bogen *Pattern Hatching — Design Patterns Applied* [Vlissides 1998], skrevet af John Vlissides. Han har selv opnået en del erfaring med brug af mønstre som nævnt i afsnit 5, og bogen er baseret på disse erfaringer.

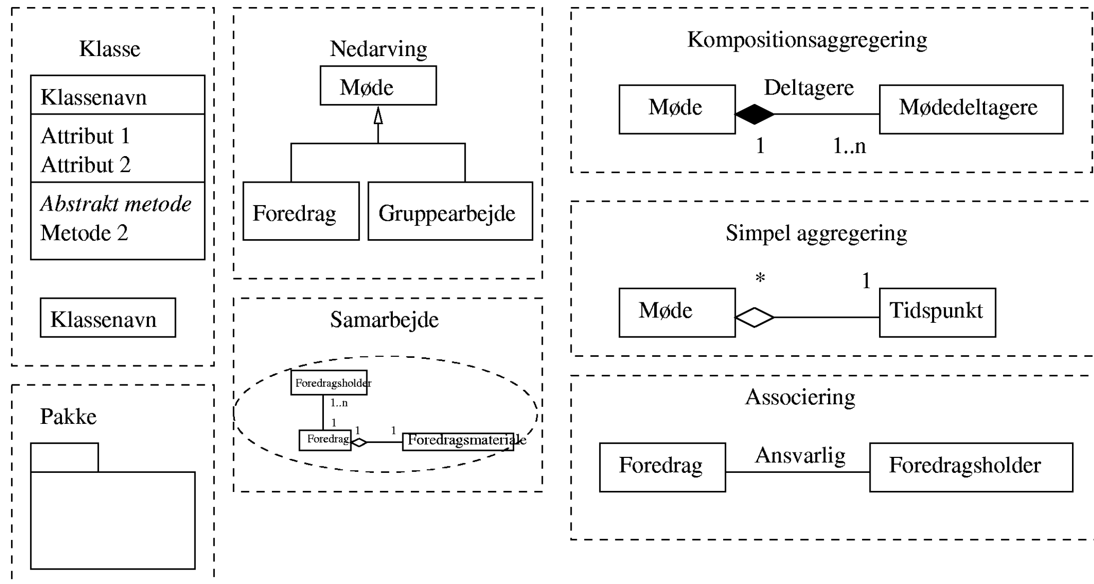
A. UML

I dette afsnit gives en kortfattet beskrivelse af den notation vi har valgt at bruge i rapporten. For illustration henvises til figur 3.

I denne notation er en *klasse* enten beskrevet ved en enkelt kasse der indeholder klassens navn, eller en tredelt kasse hvis attributterne og metoderne er vigtige for forståelsen. I sidstnævnte tilfælde indeholder den øverste del klassens navn, den midterste eventuelle variable, og den nederste de vigtigste metoder.

En klasse kan være en specialisering af en anden og være i relation til en anden på flere forskellige måder; *nedarving*, eller specialisering, af klasser er vist ved en pil med et hvidt pilehoved pegende på den klasse der specialiseres.

Kompositionsaggregering hvor det aggregerede element kun kan være en del af ét andet element, er repræsenteret ved en streg med en sort rombe med romben placeret ved den klasse der indeholder en instans af den anden. Et møde siges dermed at indeholde sine mødedeltagere idet rollen som mødedeltager eksisterer i kraft af mødet.



Figur 3 UML er en grafisk notation til beskrivelse af et system.

Dynamisk, eller *simpel, aggregering* er repræsenteret ved en streg med en hvid rombe med romben placeret ved den klasse der refererer til den anden. For eksempel er et tidspunkt ikke knyttet til ét møde; flere møder kan samtidig referere til det.

En relation kan også være uspecificeret, som en *associering,* hvor det kun angives at der er en relation, ikke hvilken slags relation. Disse relationer angives med streger, eventuelt med relationens navn angivet ved stregen.

Kardinaliteten, hvor mange instanser af en bestemt klasse der indgår i en relation, er vist med tal i begge ender af stregerne. Tallet beskriver antallet for den klasse det står nærmest; i eksemplet nedenfor beskrives ved hjælp af disse at et møde refererer til ét tidspunkt mens et tidspunkt kan referere til et vilkårligt antal møder (heri indgår tallet nul). Man kan også se det på kompositionsaggregeringen, hvor et møde indeholder et vilkårligt antal mødedeltagere (dog mindst én).

En *pakke* er en emballeret enhed, f.eks. et delsystem. Et *samarbejde* er en mængde elementer der i sammenhæng giver mere information en summen af elementerne tilsammen, f.eks. et mønster.

B. Case: Udvikling af et planlægningssystem

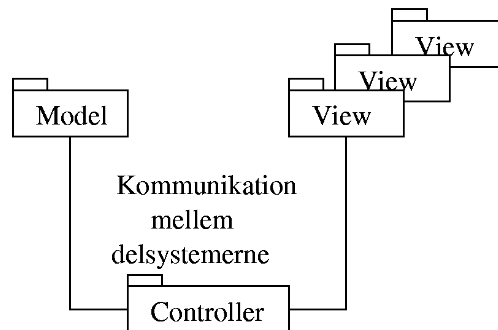
Som et eksempel på en udviklingsproces der gør brug af alle tre former for mønstre, har vi valgt et planlægningssystem til en softwarevirksomhed.

Helt overordnet skal systemet kunne holde styr på hvilke udviklere der skal til hvilke møder, og hvilke lokaler møderne skal afholdes i. Der er forskellige typer personer der skal bruge systemet, og de er interesserede i forskellige ting. Sekretærerne vil gerne kunne se det hele. Udviklerne vil gerne kunne se hvor og hvornår de skal til møder. De som sidder i lønadministrationen, vil gerne kunne se hvor mange timer udviklerne har brugt på møderne, mens de ikke er interesserede i at vide hvor møderne blev holdt.

Umiddelbart vil beskrivelsen af de forskellige syn på systemet, alt efter interesse i forskellige dele af datamængden, få en udvikler med kendskab til mønstre til at tænke på arkitekturmønsteret *Model-View-Controller,* også kaldet *MVC,* fra [Busch 1997]. Dette mønster deler et interaktivt system op i tre delsystemer som vist i figur 4.

Model-delen indeholder data og den funktionalitet der ikke er relevant for kommunikation med brugeren. Views, eller synsvinkler, viser brugeren den delmængde af data han har interesse i, på en for ham illustrativ måde. Det kan være ved tabeller, figurer eller grafer. Controlleren, eller kontroldelen, tager sig af forespørgsler fra brugerne. Kontroldelen og synsvinklerne udgør tilsammen brugergrænsefladen. Ved denne opdeling opnås en adskillelse af datamængde og brugergrænseflade. Dette er en fordel da delsystemerne i systemet på den måde kan specificeres og udvikles uafhængigt af hin-

anden. Opdelingen gør systemet mere fleksibelt m.h.t. ændringer. Brugernes synsvinkler kan ændres uden at det får indflydelse på modellen og kontroldelen; og implementationen af modeldelen kan ændres uafhængigt af kontroldelen og synsvinklerne hvis grænsefladen til kontroldelen ikke ændres.



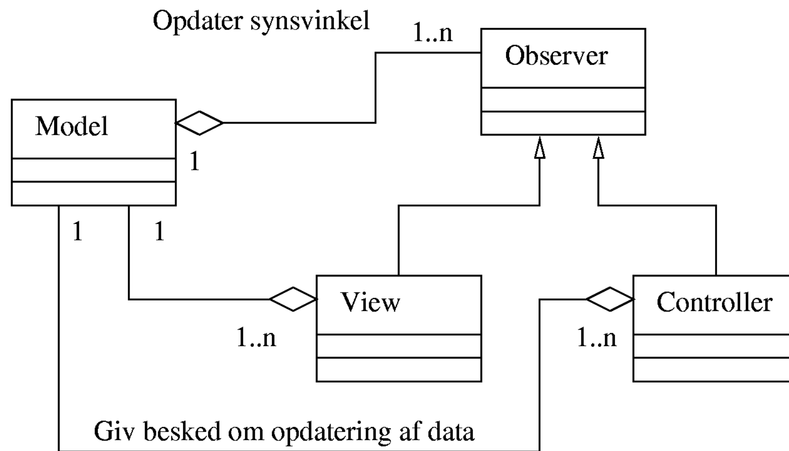
Figur 4 Model-View-Controller, et arkitekturmønster. Model indeholder datamængden og den funktionalitet der ikke er relevant for kommunikationen med brugeren. View giver fremstillingen af data til brugeren. Der kan være flere forskellige views, ét til hver slags bruger. Controller tager sig af kommunikationen med brugeren, sender de korrekte data til de implicerede views og sørger for at holde data i modellen ajour.

Med valg af arkitekturmønster (delaktiviteten „arkitekturdesign“ som vist i figur 2) er den overordnede struktur for systemet lagt fast, men der mangler stadig en stor mængde designarbejde. Den detaljerede opdeling af klasser/objekter og deres kommunikation skal findes. En stor del kommer fra analysen af problemdomænet, men der vil stadig være en del designhuller der skal fyldes ud efter den første analyse er færdiggjort. Dette kan være fordi systemet kræver mere funktionalitet end der umiddelbart kunne udledes fra analysen.

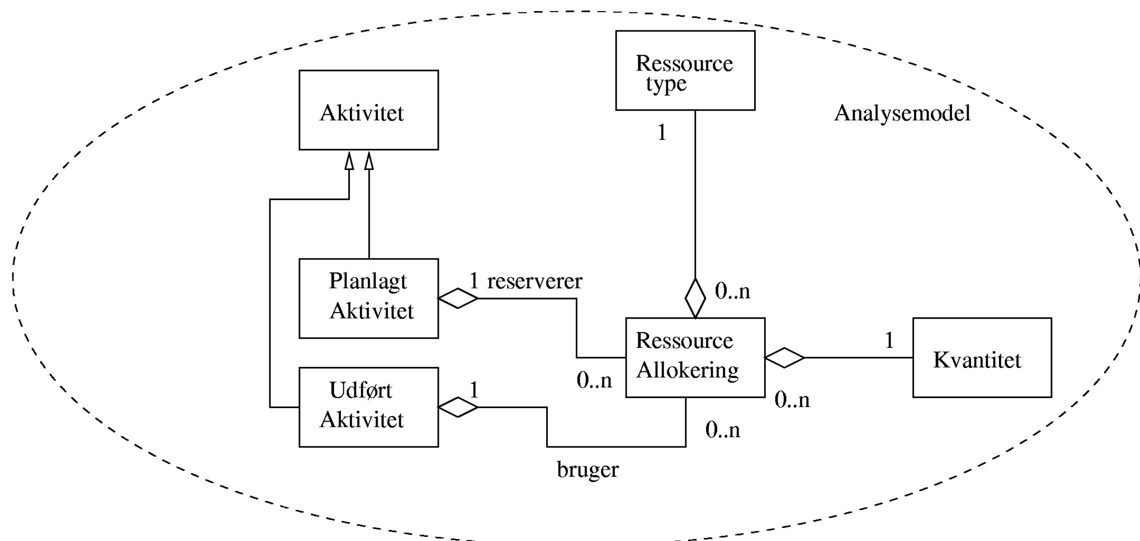
Et mønster som *MVC* beskriver den overordnede struktur, men detaljerne må tilføjes ved hjælp af mere detaljerede mønstre, designmønstre. Et åbent spørgsmål i *MVC* er hvordan og hvornår data ændres i forhold til de data der bliver ændret af en bruger fra hans synsvinkel på systemet. I en objektorienteret implementation af MVC ville man definere separate klasser for hver komponent i mønsteret. Så kan designmønsteret *Observer* fra GoF-bogen bruges til at bestemme kommunikationen mellem delsystemerne som det er vist i figur 5.

Strukturen for dette mønster er næsten den samme som for *MVC*. Fordi *Observer* er et designmønster ligger data og view nu i objekter i stedet for i delsystemer. Kontroldelen i *Observer* ligger i data- og synsobjekterne i stedet for at ligge separat. Kommunikationen mellem data og synsvinkler foregår ved at brugeren ændrer data i sin synsvinkel, denne synsvinkel sender besked til dataobjektet, som automatisk sender besked om ændringen videre til alle andre synsvinkler der er tilknyttet dataobjektet. Når dette designmønster bruges til at bestemme kommunikationen mellem delsystemerne

beskrevet af arkitekturmønsteret, vil systemet arve dets form for kommunikation og automatisk propagere alle ændringer ud til alle brugersyn.



Figur 5 En observer sørger for at kommunikationen holder data ved lige på den ønskede måde. En observer ændrer i modellen når der ændres i et view, modellen giver besked til Controller om at opdatere indholdet i alle views, og ændringen vises derefter i alle views.



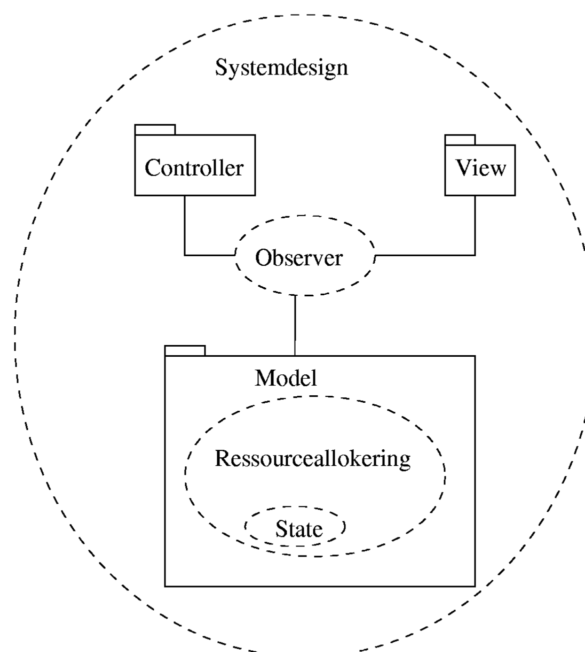
Figur 6 En aktivitet er planlagt eller udført og vil derfor enten reservere eller bruge en ressource. Ressourceallokering holder styr på hvilken type ressource der allokeres og hvor meget der er allokeret. Figuren er et eksempel på en analysemodel som vist på figur 2. Bemærk at dette er et analysemønster og at nedarvingspilene til *Aktivitet* kun betyder at *Planlagt aktivitet* og *Udført aktivitet* skal kunne behandles ens, ikke at de nødvendigvis skal være underklasser af samme superklasse.

Til analysen af problemdomænet hører en udvikling af en analysemodel (delaktiviteten analyse som vist i figur 2). Modellen skal tage hensyn til banale ting, såsom at folk ikke kan være to steder på en gang, og mere komplicerede ting som pladsbegrænsning i

lokaler alt efter hvilken slags møde det er. I denne analyse bruges et mønster fra kapitlet om planlægningsmønstre i Fowlers bog. Vi springer direkte til ressourceallokeringen da det, fra vores synspunkt, er det vigtigste at få modelleret. Som det ses af figur 6 kan en aktivitet være foreslået eller udført. Forskellen ligger i om den eller de ressourcer den allokerer, er reserverede eller brugte. Og det er netop pointen i dette mønster.

Analysemønsteret giver mulighed for at modellere flere udvidelser af denne model, men vi holder os i dette eksempel til ovenstående udgave af modellen.

Modellen vil give anledning til design der skal ligge i modeldelen af systemet (som illustreret i figur 7) da den beskriver en grundlæggende funktionalitet for systemet. Det er kontroldelen af systemet der tager sig af input fra brugeren, så hvis brugeren igennem sit syn kommunikerer at en aktivitet er udført, vil kontroldelen fortælle modeldelen om det, og modeldelen vil derefter tage sig af det ifølge sin funktionalitet.



Figur 7 Denne figur er et eksempel på indholdet af en systemdesigncirkel som vi så den i figur 2. Den viser det overordnede systemdesign beskrevet ved *MVC* hvor kommunikationen mellem delsystemerne er beskrevet ved *Observer*. Inden i modeldelen af *MVC* er analysemønsteret *ressourceallokering* brugt i analysen, og designet af dette beskrives ved *State*-mønsteret.

Idet ressourceallokering er en analysemodel, giver den ikke et design i sig selv. F.eks. kan det give god mening at lade ressourcetype og kvantitet smelte sammen til én klasse i designet for at opnå en forenkling hvis det ikke ødelægger analysemodellen. I en situation hvor der kun bruges én type ressourcer, f.eks. papir, er det kun nødvendigt at

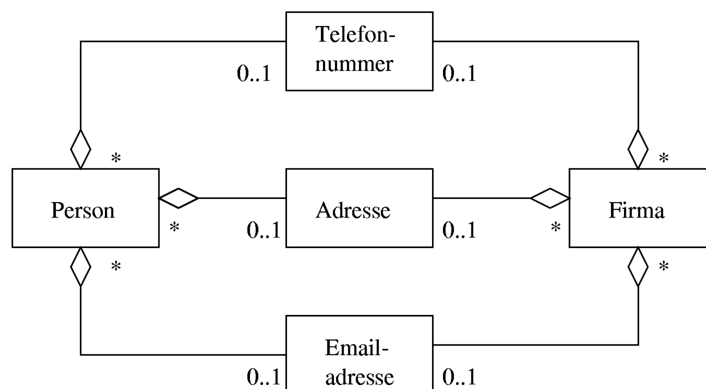
angive kvantiteten fordi det er oplagt hvilken type ressourcen har. I sådan et tilfælde ville det give et unødvendigt detaljeret design hvis analysemodellen blev fulgt til punkt og prikke. Til gengæld kan designmønsteret *State* fra GoF-bogen anvendes direkte på analysemodellen. Intentionen med *State* er at give en løsning på det designproblem der opstår når man gerne vil modellere at et objekt skifter tilstand og dermed funktionalitet igennem dets levetid, afhængigt af ydre påvirkninger (som brugerinput). I modellen i figur 6 er *aktivitet* et objekt der skifter tilstand fra *planlagt* til *udført* og dermed skifter funktionalitet, blandt andet er ressourcerne brugte i stedet for bare reserverede. Denne funktionalitet kan beskrives i designet ved hjælp at *State.*

Lad os opsummere ved hjælp af figur 7. Til skelettet af systemet bruges *MVC* med en intern kommunikation beskrevet ved *Observer.* En del af modeldelen er analyseret ved hjælp af *Ressourceallokering,* og denne analysemodel ligger til grund for et design hvori der bruges *State.*

C. Eksempler på mønstre

C.1 Analysemønstre — et eksempel

For at give et eksempel på et simpelt analysemønster vises *Party*[5]-mønsteret fra bogen „Analysis Patterns — Reusable Object Models" skrevet af Martin Fowler:



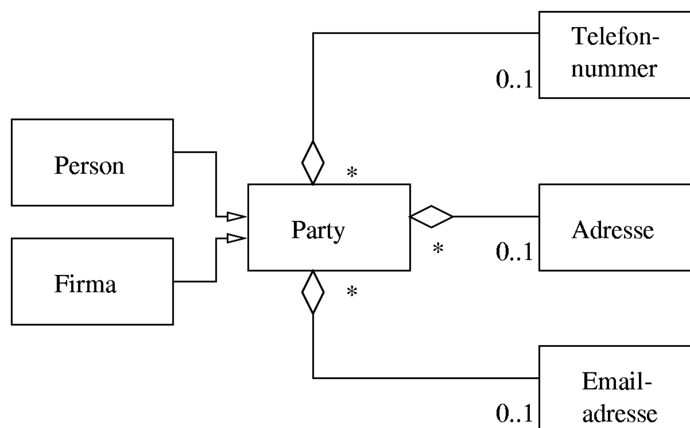
Figur 8 En model af en telefonbog hvor man skal tage hensyn til om man indsætter et telefonnummer ved en person eller et firma. Objekterne er ikke repræsenteret som klasser da det er en analysemodel og dermed ikke bestemt at de skal designes som klasser i systemet.

[5] Bemærk at „party" i denne sammenhæng ikke betyder „fest", men „part".

I hver beskrivelse af et analysemønster følges et eksempel fra den virkelige verden som læseren i større eller mindre grad kan forholde sig til, alt efter egen erfaring. I dette mønster er eksemplet dog til at følge idet Martin Fowler beskriver kompleksiteten af at modellere en privat telefonbog af den slags vi alle sammen har liggende i skuffen. Den umiddelbare model ville se ud som figur 8.

Han skriver at et af problemerne ved at modellere en indekseret telefonbog ligger i at man ofte gerne vil have oplysninger ikke bare om en person, men ind imellem også om et firma, og at det ikke er nødvendigt at tale med en bestemt person i det firma hvis man bare skal bestille en pizza. Så han leder efter en generalisering af de to begreber person og firma så han får noget at hænge oplysninger op på uden at skulle tage hensyn til om det er et firma eller en person.

Hans løsning er at lave en abstraktion over person og firma så de for alle praktiske formål opfører sig ens. Denne abstraktion kalder han „party“ og analysemodellen ser nu ud som vist i figur 9.



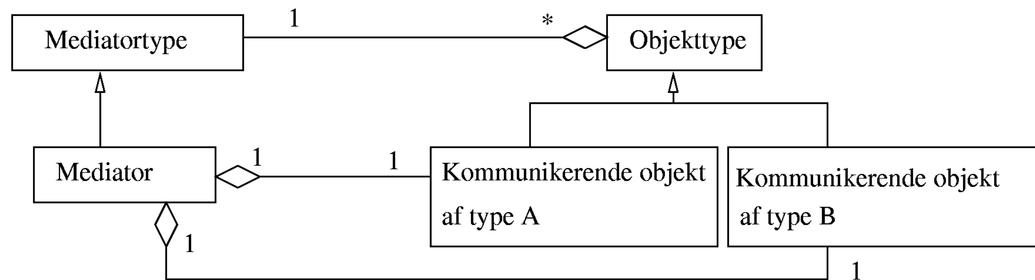
Figur 9 En model af en telefonbog hvor person og firma er smeltet sammen i en abstrakt gruppe „party“. Der gælder det samme for objektpræsentationen som i figur 8. Bemærk at dette er et analysemønster og at nedarvingspilene til *Party* kun betyder at *Person* og *Firma* skal kunne behandles ens, ikke at de nødvendigvis skal være underklasser af samme superklasse.

Dette simple analysemønster kan nu bruges i mange forskellige sammenhænge i andre analysemønstre hvor abstraktionen bliver en del af en større løsning. For at give en bedre forståelse af hvad analysemønstre er, har vi inkluderet brugen af et knap så simpelt mønster i vores eksempel i appendiks B.

C.2 Designmønstre — et eksempel

Som et eksempel har vi valgt *Mediator*-mønsteret fra GoF-bogen [Gamma 1994].
Motivationen for at have dette mønster er at det ville være nyttigt at kunne lave et fleksibelt og overskueligt design af et system der indeholder mange objekter der kommunikerer med hinanden. Objektorienteret design lægger op til at opførsel distribueres ud til de deltagende objekter. Den umiddelbare måde at lade mange objekter kommunikere med hinanden på er ved simpelthen at lade dem kende til hinanden og lade dem kommunikere direkte med hinanden. Problemet med denne løsning er at objekterne bliver stærkt forbundne og at det dermed bliver svært at overskue og ikke mindst ændre i systemet.

Designmønsterets løsning er at tilføje et ekstra objekt *Mediator*, eller mægler, som vist i figur 10.



Figur 10 En mægler tager sig af kommunikationen mellem et antal objekter i et system.

Denne mægler vil nu kende til alle de andre objekter og tage sig af kommunikationen mellem dem. Så når et objekt bliver ændret eller fjernet, kan den del af kommunikationen der havde med den at gøre, ændres i mægleren, og det vil direkte kunne ses hvilke andre objekter der var afhængige af objektet. Dette er mærkbart nemmere end at skulle søge igennem alle de andre objekter for at se hvilke der havde tilknytning til det.
Et designmønster indeholder dog mere end en beskrivelse af motivationen bag det og en struktur for designløsningen. Der er også en beskrivelse af den sammenhæng mønsteret kan bruges i, altså en samling situationer hvori det kan være fornuftigt at anvende mønsteret. I dette tilfælde når:

- der findes en mængde af objekter der kommunikerer på en veldefineret, men kompleks måde; den resulterende afhængighed imellem objekter bliver ustruktureret og svær at overskue;
- genbrug af et objekt er besværlig fordi objektet referer til og kommunikerer med mange andre objekter;
- en opførsel der er distribueret mellem flere klasser, skal kunne tilpasses uden brug mange underklasser.

En mere detaljeret viden om hvordan mønsteret skal anvendes, kan findes i afsnittene om deltagende objekter, kommunikation mellem dem og implementation. Desuden giver GoF-bogen kodeeksempler og en beskrivelse af de konsekvenser, positive som negative, der er ved anvendelsen af dette mønster.

C.3 Arkitekturmønstre — to eksempler

Vi har valgt at vise to eksempler på arkitekturmønstre fra bogen: „A System of Patterns“ [Busch 1997], *Layers* og *Broker.* Dette katalog af mønstre starter hver mønsterbeskrivelse med en kort præsentation af mønsterets brugbarhed, og for *Layers* beskrives dette som følger:

> Dette arkitekturmønster hjælper med at strukturere applikationer der kan opdeles i grupper af delopgaver hvor hver gruppe af delopgaver har sit eget abstraktionsniveau.

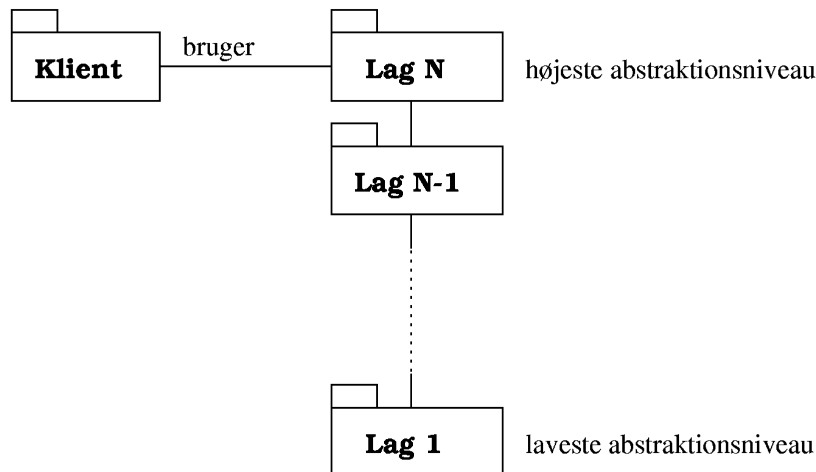
Dernæst følger et eksempel på brug af mønsteret fra virkelighedens verden. For Layers findes et velkendt eksempel: netværksprotokoller. En netværksprotokol specificerer en mængde forskellige abstraktionsniveauer, helt nede fra detaljerne om bittransmissionen til højniveau applikationslogik. Derfor bruger designere flere delprotokoller og arrangerer dem i forskellige lag. Hvert lag tager sig af et bestemt aspekt af kommunikationen og bruger den service laget under det stiller til rådighed. At der bruges en lagdeling i stedet for at have det hele i en stor blok kan have mange årsager. Det gør det nemmere at arbejde i hold og at lave skridtvis kodning og afprøvning, og det øger muligheden for genbrug af dele af systemet. Bedre implementationsteknologier såsom nye sprog eller algoritmer kan indarbejdes på en simpel måde ved at man omskriver en afgrænset mængde af koden.

Motivationen for at bruge netop dette mønster ligger i problembeskrivelsen, hvor en situation i hvilken dette mønster er en god løsning, beskrives. Problemet der kan lede til anvendelse af *Layers* er følgende: Der skal designes et system. Et fremtrædende karakteristikon ved systemet er blandingen af høj- og lav-niveau-emner. Højniveau-emnerne afhænger af lavniveau-emnerne Her betegnes lavniveau som værende på hardware-niveau mens højniveau er brugergrænseflader.

Løsningen ligger, som vist af strukturdiagrammet i figur 11, i at inddele systemet i lag med forskellige ansvarsområder. Strukturér systemet i et passende antal lag og placér dem ovenpå hinanden.

Bemærk at beskrivelsen af dette mønster ikke viser præcis hvordan disse lag skal designes. Det giver bare en begrebsmæssig synsvinkel på hvordan det kan gøres. Hovedstrukturen er at der ikke er anden kommunikation mellem lagene end mellem den der foregår mellem to lag der ligger umiddelbart efter hinanden. Dette giver den ønskede

arkitektur hvor velafgrænsede dele af systemet kan genbruges og ændres uden at berøre andre dele. Og det kan bygges af flere grupper programmører da de kan arbejde på hver sit delsystem fordi hvert delsystem har et veldefineret ansvarsområde og interface til delsystemet lige over og lige under.



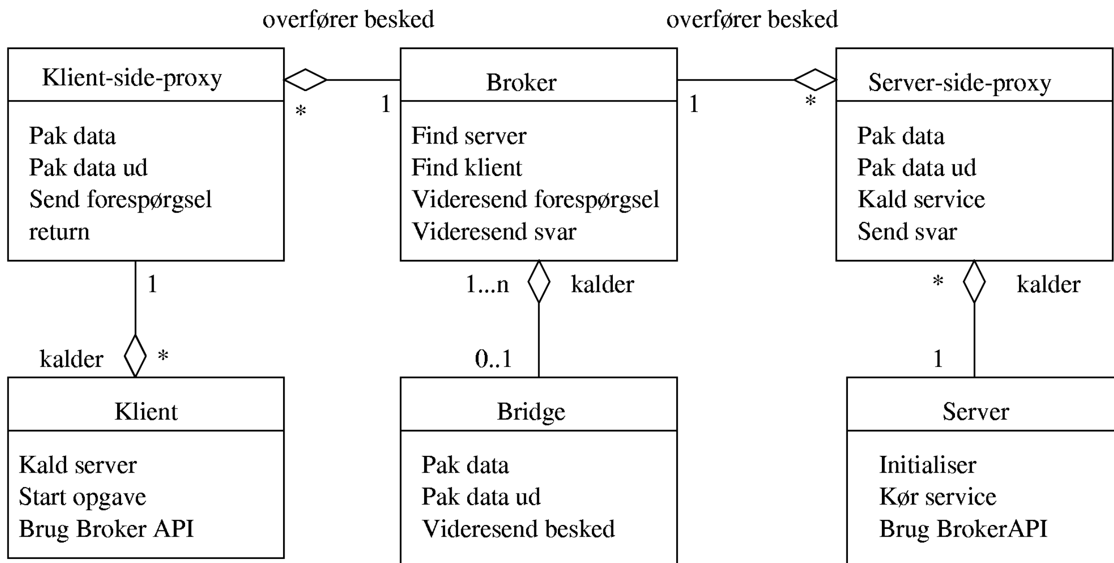
Figur 11 En lagdeling af et system til forenkling af systemudvikling og vedligeholdelse.

Et andet eksempel på et arkitekturmønster er *Broker*. Mønsteret kan bruges til at strukturere distribuerede softwaresystemer med komponenter der er placerede forskellige steder og som kommunikerer med hinanden ved hjælp af kald af hinandens metoder. En Broker-komponent er ansvarlig for at koordinere kommunikation, såsom at videresende forespørgsler, samt at sende transmissioner og exceptions. Eksemplet fra den virkelige verden er et byinformationssystem der er designet til at køre på et WAN. Nogle computere i netværket er værter for en eller flere servicefunktioner der vedligeholder information om begivenheder, restauranter, hoteller eller offentlige transportmidler. Turister kan så, fra alle steder i byen, finde de oplysninger de er interesserede i, ved hjælp af en WWW-browser. Data er distribueret over netværket i stedet for at ligge i terminalerne. Da det forventes at systemet vil ændre sig og vokse kontinuerligt, skal de individuelle servicefunktioner afkobles fra hinanden. Terminalerne skal yderligere kunne tilgå disse funktioner uden at kende lokaliteten af dem.

Det er altså et distribueret system med uafhængige sammenhængskomponenter. Motivationen for at bruge *Broker*-mønsteret ligger i at det at lave et system bestående af separate delkomponenter giver større fleksibilitet, vedligeholdelsesmuligheder og muligheder for ændringer end ét stort system. Problemet med et distribueret system af kommunikerende komponenter er dog at det bliver begrænset af den store afhængighed komponenterne imellem. Det bliver afhængigt af de kommunikationsmekanismer der bliver brugt, klienter skal vide hvor serverne ligger, og i mange tilfælde er løsningen begrænset til at blive skrevet i ét programmeringssprog.

Der ønskes altså et system der kan lade komponenterne tilgå hinandens funktionalitet uden at vide hvor de andre komponenter ligger, hvor der kan tilføjes, skiftes eller fjernes komponenter på runtime, og hvor arkitekturen gemmer system- og implementations-afhængige detaljer fra brugerne af komponenter og funktioner.

Løsningen er at introducere en formidler-komponent, broker, for bedre at kunne separere klienter og servere. Som det ses af figur 12, indeholder strukturen af *Broker*-mønsteret følgende samarbejdende komponenter: klienter, servere, brokere, bridges, klient-side-proxier og server-side-proxier.



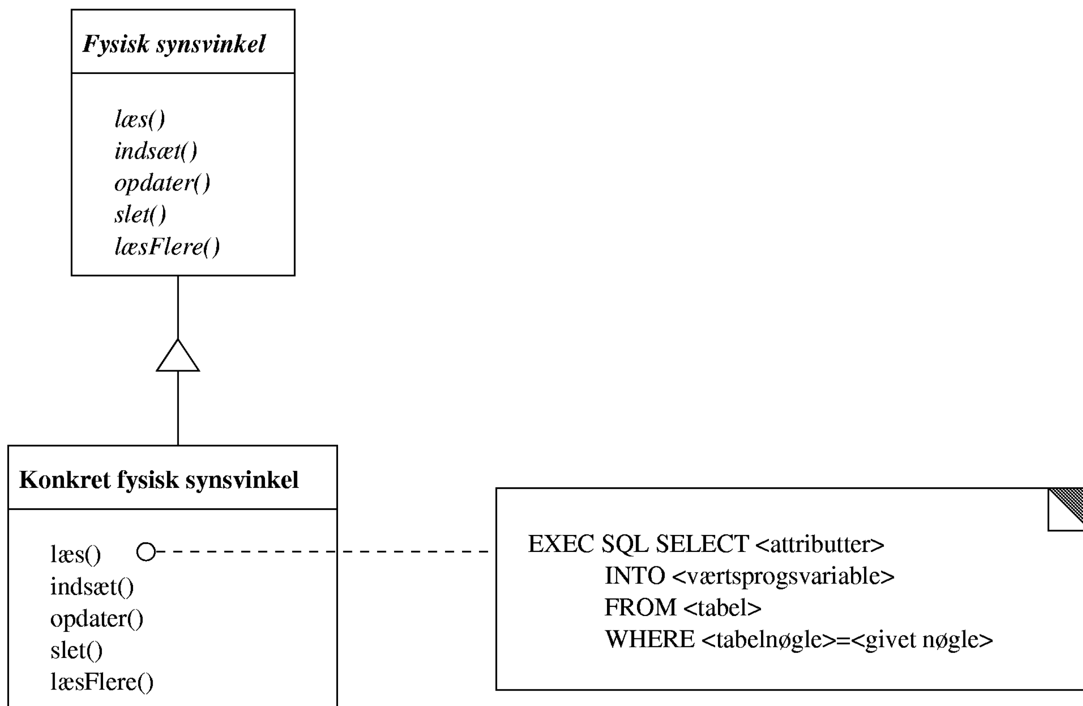
Figur 12 En kommunikationsprotokol mellem servere og klienter med forskellige fysiske placeringer.

Servere registrerer sig hos brokeren. Klienter tilgår servernes funktionalitet ved at sende forespørgsler via brokeren. En brokers ansvarsområde er at lokalisere serveren, videre-sende forespørgslen til den og transmittere resultater og exceptions tilbage til klienten. Klient-side-proxier repræsenterer et lag mellem klienter og brokeren; for at gemme kommunikationsdetaljerne mellem broker og klient for klienten. Analogt for server-side-proxier. Bridges er komponenter der tager sig af kommunikationsdetaljerne mellem forskellige brokere. Bridges er naturligvis kun nødvendige hvis der skal bruges mere end én broker eller hvis kommunikationen mellem dem er givet af en bestemt standard, som for eksempel http eller ftp. Eksempler på brug af *Broker* er Common Object Request Broker Architecture (CORBA), OLE 2.x og World Wide Web.

C.4 Databasemønstre — 2 eksempler

Vi præsenterer i dette afsnit et eksempel fra [Chasms 1995] og ét fra [Keller 1996]. Da det sidste er det enkleste, tager vi det først.

Fra [Keller 1996] har vi valgt mønsteret *Fysiske views* (Physical Views). Dette mønster anviser en indkapsling af databasen på forholdsvis lavt niveau. (Ovenpå de fysiske views lægger [Keller 1996] logiske views, evt. adskilt af et forespørgselsmellemled (*Query Broker;* beslægtet med brokeren fra forrige afsnit, men ikke den samme).) Indkapslingen gør det muligt at benytte databaseoptimeringstricks såsom denormalisering, overløbstabeller og kontrolleret redundans uden at applikationsprogrammer der bruger de fysiske views, berøres; kun implementationen af det eller de relevante fysiske views skal ændres.



Figur Error! Unknown switch argument.**3** Strukturen for databasemønsteret *Fysiske views.*

Mønsterets løsning er at indkapsle hver tabel og hvert view i databasen med et objekt der kaldes et fysisk view, som vist i figur 13. (En såkaldt overløbstabel indkapsles sammen med sin „forældretabel“.) De fysiske views har en fælles superklasse der erklærer de fælles operationer (f.eks. læs, indsæt, opdater, slet og læsFlere). Man kan definere yderligere fysiske views efter behov, f.eks. hen over joins. Da fysiske views er generiske, anbefaler mønsteret at man bruger en generator til at fremstille dem — eller en første udgave af dem. De kan evt. implementeres ved hjælp af „stored procedures“.

Fysiske views kan også bruges til at indkapsle ikke-relationelle databaser, flade filer m.v.

Mønsteret *Broker* fra gruppen af dynamiske mønstre i [Chasms 1995] er nok det mest centrale mønster i den artikel. Det anviser også en indkapsling af databasens tabeller. Enhver tabel indkapsles her af en databasebroker der ikke kun tager sig af tilgang til tabellen, men også af at konstruere objekter af de data der hentes fra databasen. Ligeledes accepterer den at tage imod objekter hvis dataindhold skal gemmes i eller slettes fra databasen. En brokerklasse kan evt. parametriseres med objekttypen således at fælles operationer kun specificeres ét sted. Brokeren kan evt. indeholde og styre en cache; hvis der er mulighed for at brokeren flere gange bliver bedt om at hente de samme data, vil det ofte være vigtigt at den ikke blot genererer flere objekter med det samme indhold, men returnerer en reference til *det samme objekt* hver gang. Ifølge [Chasms 1995] kan brokeren også styre transaktioner overfor databasen. Dette mønster er altså et andet end det arkitekturmønster af samme navn der blev præsenteret i afsnit C.3, og også et andet end mønsteret Query Broker fra [Keller 1996].

D. Litteraturliste

[Alex 1977] Christopher Alexander o.a.: *A Pattern Language*
Oxford University Press, 1977

[Alex 1979] Christopher Alexander: *The Timeless Way of Building*
Oxford University Press, 1979

[Rum 1998] Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language Reference Manual* , Addison-Wesley, 1998

[Chasms 1995] Kyle Brown & Bruce G. Whitenack:
Crossing Chasms — A Pattern Language for Object-RDBMS Integration.
http://members.aol.com/kgb1001001/Chasms.htm
ftp://members.aol.com/kgb1001001/Chasms/chasms.pdf

[Chasms 1998] Kyle Brown: *Crossing Chasms: The architectural patterns.*
http://www.ksccary.com/Articles/ArchitectualPatterns/ArchitecturalPatterns.htm

[Busch 1997] F. Buschmann o.a: *Pattern-Oriented Software Architecture: A System of Patterns*
John Wiley & Sons, 1997

[PloP] James O. Coplien and Douglas C. Schmidt (editors):
Pattern Languages of Program Design 1-3
Addison-Wesley, 1995-1997

[Fowler 1997] M. Fowler: *Analysis Patterns — Reusable object models.*
Addison-Wesley,1997

[Gamma 1994] E. Gamma, R. Helm, R. Johnson & J. Vlissides:
Design Patterns: Elements of Reusable Object-Oriented Software
(Også kaldet GoF-bogen) Reading, MA: Addison-Wesley, 1994

[Keller 1996] Wolfgang Keller: *A Design Cookbook for Business Information Systems*
http://www.sdm.de/g/arcus/cookbook/relzs/index.htm

[Mat 1997] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen og J. Stage:
Objektorienteret Analyse og Design
Forlaget Marko, Ålborg, 1997

[Mow 1998] T. Mowbray o.a.: *Anti Patterns: Refactoring Software*
John Wiley & Sons, 1998

[Rum 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen:
Object-Oriented Modeling and Design
Prentice Hall, 1991

[Vlissides 1998] John Vlissides: *Pattern Hatching — Design Patterns Applied*
Addison-Wesley,1998

[uiuc1999] Mønstrenes hjemmeside:
http://st-www.cs.uiuc.edu/users/patterns/patterns.html

[oopsla87] De første mønstre:
http://c2.com/doc/oopsla87.html

[publication1997] Bøger om mønstre:
http://www.sigs.com/publications/docs/objm/9703/9703.patterns.html

[ICSE1996] Beskrivelse af erfaringer med brug af mønstre:
http://www.bell-labs.com/~cope/Patterns/ICSE96/icse.html