

Indholdsfortegnelse

XML (eXtensible Markup Language) (Y)	1
Transaktioner (Y)	3
Objektorienterede databaser (Y)	6
Relationelle model (fork. RM) og SQL (Y)	8
UML (Unified Modeling Language) (Y)	10
Tilstandsdiagrammer (Y)	12
Datamodeller (Y)	13
Use-case og sekvensdiagram (Y)	17
Temporale modeller (Y)	19
Objekt orienterets fordele/ulemper (Y)	21
Datadistribution (Y)	22
Teknisk arkitektur (Y)	25
Klassediagram & (Y)	28
Mønstre (Y)	29

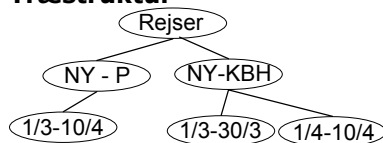
XML (eXtensible Markup Language) (Y)

- **Struktur**

- Generelt

- **Bygger på semistrukturerede data**
 - **Et markup language som HTML**
 - **Dets tags har ingen forskrevne regler**
 - **nøglen til XML's rolle i datarepræsentation- og udveksling.** tilvejebringer indholdet af hver værdi samt bevare betydningen af hver værdi `<X> tekst </X>`
 - **Tags gør beskeden selv-dokumenterende**
 - **Kan tilføje nye tags**
 - Et tag kan have attributter `<navn attribut="xxx"></navn>`, bruges ofte til data som ik skal vises i udprintningen
 - Namespaces
 - globale unikke navne
 - Nødvendige ved udveksling ml. applikationer
 - Ofte en hjemmesideadresse der er specifik
 - [CDATA]
 - Bruges til at gemme data som indeholder værdier hvori der bl.a. er tags, der ik skal tolkes i xml

- **Træstruktur**



```
<rejser>
  <rejse navn="NY-P">
    <periode>1/3-10/4</periode>
  </rejse>
  <rejse navn="NY-KBH">
    <periode>1/3-1/4</periode>
    <periode>1/4-10/4</periode>
  </rejse>
</rejser>
```

- **Velformed**

- **Træstruktur**
 - **Indlejrede data**
 - `< > .. </ >`

- **Anvendelse**

- **Udveksle data ml. to systemer**

- Sende rejseplaner til brugere
 - Kræver ikke kendskab til programmer hos parterne
 - Kan sende via en fast standard (Scheme)

- **Udtrække data fra flere systemer**

- Fra flere kilder og vise dem på en hjemmeside
 - Det kunne være en personal financial manager der trækker alle ens konti fra forskellige banker og lader dig administrere disse centralt.

- **XML-relaterede sprog**

- **Schema (Valide)**

Anvendes til at begrænse den information der kan blive lagret i databasen og til at begrænse data typerne af den lagrede information.

- **DTD**

- **en liste af regler for hvilket mønster af subelementer der fremgår inden for et element**
 - a specification that accompanies a document and identifies what the funny little codes (or markup) are that separate paragraphs, identify topic headings, and so forth and how each is to be processed
 - any location that has a DTD "reader" (or "SGML compiler") will be able to process the document and display or print it as intended
 - Eks. HTML vist via en compiler som en browser
 - Problemer
 - Da rækkefølgen ikke er ligegyldig er det svært at deklarerer en DTD, hvis man har en mængde uordnede data.

- (SAX)
 - **XML Schema**

- This description can be used to verify that each item of content in a document adheres to the description of the element in which the content is to be placed.
 - presents the interrelationship between the attributes and elements of an XML object
 - To create a schema for a document, you analyze its structure, defining each structural element as you encounter it
 - Nyeste og bedste
 - Mere direkte
 - Skrevet i XML
 - Kræver derfor ikke en parser
 - Selvdokumenterende
 - Kan blive forespurgt gennem XSLT (XML Transformations)
 - Tillader brugerdefinerede typer
 - Komplekse typer kan udvides ved en form for nedarvning
 - Den tillader unikhed og fremmednøgle begrænsninger
 - Nogle ser det dog som unødvendig kompleks
- **Quering and Transformation**
 - er essentielle til at udtrække information fra store 'bodies' af XML data, og til at konvertere data mellem forskellige repræsentationer (schemas) i XML
 - Sprogene benytter træstrukturer
 - **Xpath**
 - sprog til at udtrykke stier
 - Kan skrive stier i stil med det man gør i dos ☺
 - **XSL**
 - Skabe præsentationer og kan bruges til at gennemgå træstrukturen både med if sætninger, loops m.v.
 - **XSLT**
 - Kan transformere et XML dokument til et andet, altså derved gå fra eks. XML til XHTML m.v. Kan også bruges til at forespørge, og derved udvælge specifikke data
 - Hørte oprindeligt under XSL som er et style sheet sprog til XML
 - **Xquery**
 - for forespørgsler i XML data
 - Minder om SQL
 - Joins
 - Where
 - Distinct
- **Application Program Interface (API)**
 - er forskrevet metode i en applikationen, der kan anvendes til at tilgå andre applikationer. Eks. at bruge JAVA men stadig have adgang til XML
 - **DOM (Dokument object model)**
 - Manipulationssprog der ser træstrukturen som et objekt og har en række metoder til gå gennem et træ, og til bl.a. at kunne gemme ting
 - SAX
- Datalagring
 - Gemme i relationel DB og udtrække data
 - Minder dog mest om den hierarkiske datamodel
 - Gemme i fladfil
 - Gemme i XML db, kan bygges ovenpå relationelle DB'ere
- **Fordele**
 - **Middel til kommunikation**
 - **Web**
 - **B2B**
 - **Wap**
 - Bredt accepteret standard
 - **Kan nemt udvides med nye tags**
 - **Tags giver mening**
 - **Ved at blive den dominerede standard til data udveksling**
 - Undgår joins pga. nested data
- **Ulemper**
 - Kan ikke helt sammenlignes med at gemme data i databaser da man bl.a. gentager tags meget
 - **Redundante data**

Transaktioner (Y)

- **Definition**
 - **Enhed af programudførelse (program execution), der accesser og muligvis opdaterer forskellige dataenheder**
- **Årsag**
 - Fokus er på at holde **databasen konsistent** (Sammenhæng i data før og efter, data er valide), før og efter en transaktion
 - Fejl af forskellige typer, såsom hardwarefejl og system **crashes**
 - **Samtidig transaktionsudførelse** (concurrent execution) Ønske om parallelisme -
 - → Øget processor og disk brug
 - → Reduceret gns. Responstid
 - → Mulighed for problemer med inkonsistens og deadlocks
 - Sikres gennem ACID
- **Transaktionseksempel** (i relation til vores opgave, så kunne vi forestille os, at A er en kundes konto hos VIPair, og B udligning på en "betalingskonto" -> meget forsimplet)
 1. read(A)
 2. A := A - 50
 3. write(A)
 4. read(B)
 5. B := B + 50
 6. write(B)
- **Transaktionsegenskaber: ACID** (bruger eksemplet til at forklare)

	Definitioner	Til eksemplet	Centraliserede databaser	Distribuerede databaser	Sikres via
Atomicity	Enten skal alle eller ingen operationer i transaktionen reflekteres i databasen	A + B = uændret efter transaktions-udførelse	(ATOMICITY) Transaktionslog / write ahead log (skal kunne lave roll-back, hvis ikke man laver commit)	(ATOMICITY) 2 fase commit:	Database systemets recovery-management
Consistency (Konsistens)	Udførelsen af transaktionen efterlader databasen i en konsistent tilstand	fejler efter 3 og før 6, → transaktions-opdateringen ikke blive reflekteret i db → ellers inkonsistent	(Serialiserbar) 2 fase låsning (kun 1 der skriver)	(Serialiserbar) 2 fase låsning virker næsten... så derfor: + distribueret deadlock detection	Concurrency control schemes.
Isolation	Transaktionen er isoleret og udføres enkeltvis. Samt har intet kendskab til hinanden	En anden transaktion ser transaktionen ml. trin 3 og 6 → inkonsistent db. Undgå → serielle transaktioner, samtidigt dog store fordele.			
Durability (Holdbarhed) (Persistens)	Når transaktion er fuldført og DB er konsistent, er data lagret og kan klare et crash	Når transaktionen er foretaget må der på ingen måde ske fejl som medfører at den ikke er reflekteret i databasen mere	(Persistens) Disk: - Backup + Transaktionslog - Mirror ("replikering")	(Persistens) Det samme + Replikering (ægte) fra central DB Vi har valgt mirror, men ved nærmere eftertanke, så vil periodevis backup være billigere til mindre kritiske data	Database systemets recovery-management (Ordentlige diske Backup Mirror)

Vi har måske skrevet noget sludder om adgangen til secondarycopy i vores opgave

- Hvor er parallelisme et problem:
 - Kun læsning/read – ingen problemer – i vores opgave (og generelt) skal man dog være opmærksom på, at der ikke må læses på objekter, som har skrivelås (kommer senere)
 - Læsning og skrivning, men på forskellige objekter, fx 2 forskellige rute-afgange – ingen problemer
 - Læsning og skrivning på samme elementer => (problem)/udfordring
- **Låsning** (løsning på ovenstående)
 - **Årsag**

- **At udføre transaktioner samtidigt**

- **Låse kan være på:** (jeg har selv fundet på den for ODB)



Vi har valgt låsning på objektniveau i opgaven

- **Låse**

En mekanisme til at kontrollere samtidig adgang til et data element.

Hvis en lås ikke kan gives, må transaktionen vente indtil alle inkompatible låse er fjernet, hvorefter der gives tilladelse til transaktionen.

- **Read-lås** (s-lock, shared)
 - kan være mange read-låse på samme element (markeres at de læses)
- **Write-lås** (x-lock, exclusive)
 - Kan kun sættes hvis der ikke er andre låse sat på elementet
 - Dog kan egen readlås opgraderes til writelås

- Låse kan sammenlignes med det der sker på netværk

- Ethernet (på vej ind – pga. pris/ydelse) - Kollision – svarer til parallelisme
- Token ring (på vej ud) – Staffeten svarer til låsen

- **Problemer**

- **Deadlock**

- To transaktioner med tilhørende låse, der medfører at ingen af transaktionerne kan fortsætte og derfor må begge transaktionerne "rulles tilbage". Kan ikke undgås.
- T_1 venter på en dataenhed som T_2 har osv. og den sidste venter på en som T_1 har

- **Starvation**

- Sker ved dårlige concurrency control systems, en række read låse som gives mens at en anden venter på at få en write-lås adgang til samme data element og derfor må rulles tilbage flere gange.
 - kan undgås ved at en ny read-lås undersøger om der er nogle konflikter i lås-typerne, altså at der er en ventende write-lås.

- **Låsemetoder**

- **Two-phase**

En protokol der sikrer konflikt-serializable schedules er Two-phase locking protocol

- Fase 1: Growing Phase
 - Transaktioner kan få låse
 - Transaktioner kan ikke frigive låse
- Fase 2: Shrinking Phase
 - Transaktioner kan frigive låse
 - Transaktioner kan ikke få låse

- Protokollen sikrer serializability ("*efterfølger hinanden*"). Det kan bevises at transaktioner kan blive serialized gennem der lock point orden. Derved skulle starvation undgås men ikke deadlocks. (Serialielt – resultatmæssigt ok, men dårlig performance. Transaktioner skal gerne have parallelismens ydelses fordel, men samtidig gøres serialiserbar)

- **Transaktionspunkter i opgaven**

- Reservation

- Flere bestiller prøver at bestille samme plads
- Vigtigt: Hvornår den er reserveret
- Mange transaktioner → Parallelisme, især ved skrivning
- Flere skriver samtdigt → Flere tror de bestiller plads
- 2 Fase låsning
 - Deadlocks

- Betaling

- 2 reservationer samtidigt og på samme debitor-konto, kan medføre at man kun får trukket den sidste fra debitor-kontoen.

Noget vi skal ind på???????????????

ATOMICITY, SERIALISERBAR, PERSISTENS

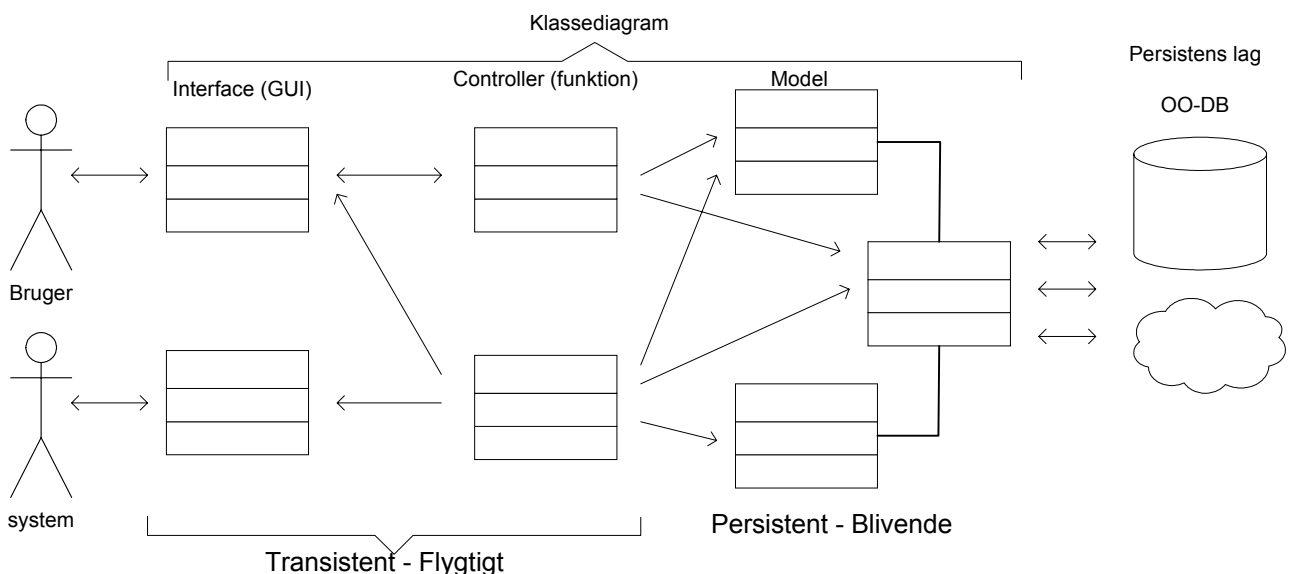
- Transaktionstilstande
 - Active = Initiale tilstand og tilstand mens udførelse
 - Partially committed = Efter at den sidste statement er udført (Distribuerede db 'ere)
 - Failed
 - Aborted = Efter at transaktionen har sat databasen tilbage til tilstanden før transaktionen ("rolled back")
 - Committed = Efter succesfuld udførelse
- Opnåelse af ACID
 - Atomicity (og durability) → Database systemets recovery-management
 - Alle opdateringer foretages på shadow kopien af databasen og db_pointeren peger først på den opdaterede version af shadow kopien når transaktionen er partially committed og alle opdaterede sider er blevet overført til disken
 - Hvis en transaktion fejler, kan den gamle konsistente som db_pointer peger på bruges og shadow kopien kan slettes.
 - Shadow database scheme forudsætter at diskene ikke fejler, og ikke optimal for store databaser, da der jo faktisk er tale om at kopiere hele databasen.
 - Consistency
 - Schedules er sekvenser der indikere den kronologiske orden hvormed instruktionerne af concurrent transaktioner udføres.
 - Schedule skal indeholde samtlige instruktioner for transaktionerne samtidig med at den må bevare ordenen ift. hvordan instruktioner optræder i den enkelte transaktion.
- Isolation
- Durability (se atomicity) → Concurrency control schemes
 - Sikre god kvalitet disk
 - Hvis cracked
 - mirror / kopi
 - periodevis backup, og transfer logs

T ₁	T ₂
read(A)	
A := A - 50	
write(A)	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
read(B)	
B := B + 50	
write(B)	
	read(B)
	B := B + temp
	write(B)

Objektorienterede databaser (Y)

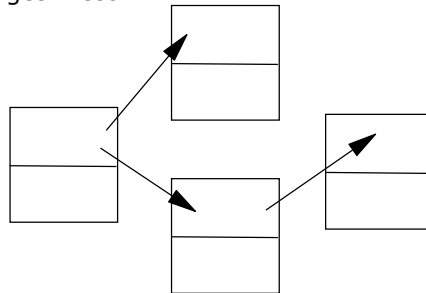
- **Objekt orienteret datamodel**

- Årsag
 - Behov for at kunne håndtere komplekse typer
 - Eks. at se en adresse som en **kompositet attribut**, i stedet for opdelt i attributter som gade, by, postnr. m.v.
 - Mister ikke relationerne mellem disse data
 - Medfører simple forespørgsler
 - Kunne gemme flere tlf.-numre i en **multivalued attribut**
 - Gemme det samlet
 - RDBMS
 - Fokus på 1970'erne: Large numbers of similarly structured data items, all having the same size. Fixed-length, small records, in 1NF
 - OODB
 - Fordele ved CAD, Case, Multimedia DB
 - Udviklet til at dække mange af de behov som nye typer af nye typer af applikationer har.
 - En tilvænnning af det object-orienterede programmeringsparadigme til database systemer. Den bygger på konceptet at indkapsle data og koden
 - Behov for at kunne håndtere objekt orienteret begreber som nedarvning
- Som er datamodellen bygger på entiteter og entitetstyper bygger oo datamodellen på objekter og klasser.
- Man bør derfor kunne sige at en oo datamodel svarer til et klassediagram som gemmes i en db
 - Objekt
 - Attributter
 - Metoder (Taler her også om **meddelelser**)
 - Klasser
 - En struktur der forklarer objekter
 - En klasse der instancieres bliver til et objekt
 - Objekt identitet
 - Dette skyldes at hvor en entitet kan bevare sin identitet, selvom der ændres nogle af sine egenskaber, så kan et objekt stadig bevare sin identitet selvom alle dets værdier, variable eller definitionerne på dets metoder ændrer sig.
 - Er unikke
 - Kan gemmes som en "object identifier" (automatisk genereret), normalen
 - Fastlægges eksternt (cpr-nr)
 - Kan gemmes i andre objekter
 - Gemmes i et andet objekt, således denne har en reference til det pågældende "object identifier"s objekt



- **Persistens**

- Fokuserer på, hvornår man gør et objekt persistent, således den lagres i databasen
- En række muligheder man kan benytte
 - By class
Man siger til en klasse at alle dets objekter skal være persistente
 - By creation
Jeg vælger ved oprettelse om klassen skal være persistent
 - By marking
Jeg vælger når jeg vil om klassen skal være persistent
 - By Reachability
Alle andre objekter som peges på i andre klasser bliver også persistente hvis "hovedklassen" er persistent
 - Bruges mest



Defineret persistent Disse bliver også persistent

- **OQL**

- Et sprog der minder meget om SQL
- Dækker det problem der er med at skulle spørge på en værdi
 - Selvom man reelt set ikke må gøre dette
 - Dette er vist en væsentlig årsag til at objekt orienteret DB ikke er gode, men som sagt man har indlagt OQL og det forbedrer det.

EKSEMPEL

- **Objekt orienteret-databaser**

- Ægte DB'ere
 - O₂ og Poet
- Relationelle med objektoverbygning
 - Oracle 9i, sybase, MySQL, DB2

<http://knuth.luther.edu/~willwalt/Public/Pub4302/Lect0514.html>

Relationelle model (fork. RM) og SQL (Y)

- Hvad er den RM
 - En relationel database indeholder en samling relationer
 - En relation er (Ekstension¹):

Kundenr	Navn	Adresse	Land
00123	Gudrun	Æblevej 12	DK
23421	Karl	Stjernegade 17	DK

Overskrifter = Attributter
 Tuple

- Relationsschema (Intension²)
 - $R=(\text{Kundenr}, \text{Navn}, \text{Adresse}, \text{Land})$.
 - $r(R)$ er således en relation af relationsschema
- Tuple: repræsenterer en relation mellem et sæt af værdier
 - Indeholder værdier (v)
 - Skal være singlevalued
 - Ikke multivalued (Flere ens type værdier i en attribut) eller composite (Består af flere forskellige værdier)
- Nøgler

Bruges til at unikt at identificere tupler i en relation

 - Supernøgle
 - 1 eller flere attributter udgør nøgle. F.eks. cpr, navn, postnr
 - Sammensat nøgle
 - 2 eller flere attributter udgør nøgle – unique
 - Candidate key / nøgle
 - Minimal supernøgle f.eks. cpr.
 - Primær nøgle
 - En valgt candidate key
 - Fremmednøgle
 - Er ikke nødvendigvis en supernøgle
 - Er ikke en nøgle, da den ikke kan udpege en tuple unikt i den relation den tilhører
- Forespørgsler**
 - Forespørgselsprog (HVORDAN)**

Er sprog hvorfra brugeren kan forespørge efter information fra databasen. Rene sprog danner grundlag for forespørgselsprog:

 - Relationel Algebra (Procedural sprog)
 - SKAL DET UDVIDES**
 - Tuple Relationel Calculus (Non-procedural sprog)
 - Domain Relationel Calculus (Non-procedural sprog).
 - SQL (HVAD) (Deklarativt sprog)
 - Nuværende udgave er SQL-92, men der er kommet en ny kaldet SQL:1999

	SQL	RA	Forklaring
Eksempel	SELECT relationB.attributA, relationA.attributB FROM relationA innerjoin relationB on relationA.attributC = relationB.attributC Where relationA.attributD = "XXX"	$\pi_{\text{attributA}, \text{attributB}}$ $(\sigma_{\text{attributD}="Værdi"}(\text{relationA}) \bowtie \text{RelationB})$	Udvælg alle tupler fra relation A join B hvor Vis attributterne attributA fra relation B og attributB fra relation A
Udvælgelse af attributter	1. Select	1. π = Project	
Udvælgelse af tupler	3. Where	2. σ = Select	
Relationer og joins	2. From	3. \bowtie = Natural-join	

- Kan bruges til at forespørge om data
- Ændre, slette, indsætte tupler
- Data-Definition Language kan bruges til at ændre, slette, indsætte relationer

- Fordele

¹ Ekstention er virkeligheden (i DB) og her ser man de enkelte entiteter/objekter

² intension er modellen som vi tegner den (noget generelt/abstrakt)

- Primære model til kommercielle data-processerings applikationer
- SimPLICITET
 - i forhold til netværksmodellen eller den hierarkiske model.
- Ulemper
 - For simpel til komplekse systemer
 - Data analyse bedre med DW

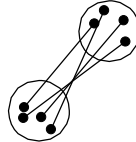
Forskellen på ER-Model og Relationel:
HØRER IKKE TIL EMNET

- ER er konceptuel, handler om struktur, man laver ingen forspørgsler, og har kun en intension, extensionen vist nedenunder er ikke en reel extension. ER-modellen er på et højere plan en den relationelle og kan også bruges til bl.a. netværksdatabasen og den hierarkiske database.

- Intension:



- Extension (hvis man kan kalde det det):



- Relationel model er på et lavere niveau og har en ekstension

I Praksis laver man intentionen i ER og implementere Extentionen i relationen

UML (Unified Modeling Language) (Y)

Hvorfor I det hele taget tale om modelleringsprog

Vi bygger modeller af komplekse systemer fordi vi ikke kan overskue dem i deres helhed. Som kompleksiteten øges, øges også behovet for modeller.

Hvem/hvorfra?

UML blev opfundet af 3 OO-guruer midt i 90'erne:

- Jacobsen, skandinavisk skole, OOA-ekspert
- Booch, amerikansk skole, OOD-ekspert
- Rumbaugh, amerikansk skole, OO Metode-ekspert

Standard: Standardiseringskomitéen OMG (Object Management Group) gør UML til en standard

Hvad + (hvorfor)?

UML er et værktøj med mange komponenter til grafisk at modellere mange aspekter i et informationssystem. Hænger meget tæt sammen med OOAD-udviklingsmetoden!

- Kravspecifikation
 - **Use-cases**
- Analyse
 - **Udvidede use-cases**
 - **Klassediagram** (kernen i UML)
 - Formål: Modellerer objekter (hvor man i ER modellerer entiteter) mht. klasser (typer), attributter og metoder
 - Struktur er vigtigt → Fokus på: analyseklasser, objekter og associeringer
 - **Objektdiagram** (intension af klassediagrammet)
 - **Black-box sekvensdiagram** (aktør vs. :system)
- Design
 - **Klassediagram** (udvidet/komplet)
 - Detaljer er vigtigt → Fokus på: metoder, attributter og specialklasser (UI, controller, container mv.)
 - **Sekvensdiagram**
 - Formål: Vise forløbet af en use-case gennem de forskellige klasser via metoder.
 - Altså tæt sammenhæng med udvidede use-cases og klassediagram i design.
 - **Tilstandsdiagram** (state diagram)
 - Viser objektets livscyklus og mulige tilstande
 - Bygger på klassediagrammet
 - Alternativt kan man bruge collaborations-diagrammer, men Klaus kan bedre lide state diagrams
 - **Deploymentdiagram**
 - Hører til modellering af teknisk arkitektur, dvs. servere, klienter og DB.

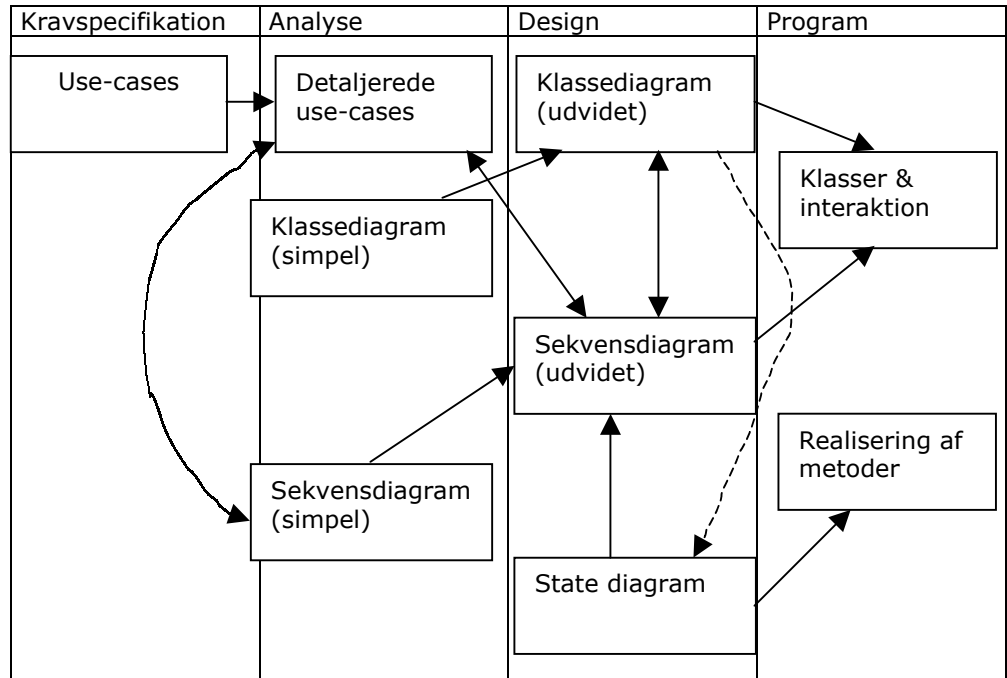
Fordele ved UML

- En **standard**, så meget udbredt → **Letter kommunikation** udviklere imellem + evt. også til klienter. Specielt use-cases er en letforståelig grafisk repræsentation af funktionelle krav.
- **Omfattende** modelleringsværktøj → dækker en stor del af systemudviklingsprocessen.
- Stor "funktionalitet" og mange **muligheder**, f.eks. associeringer, mønstre, nedarving → kan anvendes til at opbygge **komplekse** systemer.
- Stærk **sammenhæng** mellem de forskellige diagrammer → sikrer konsistens + mindsker arbejde
- Avancerede **CASE**-værktøjer → letter udviklingen generelt og programmeringen pga. den store sammenhæng i værktøjerne. Sikrer desuden **konsistens** i modellerne.
- Skaber grundlag for en omfattende **dokumentation**

Ulemper ved UML

- **Sværere at forstå og anvende** end f.eks. ER
- Det kan ikke svare sig at anvende ved **mindre systemer**
- Lægger sig op af **OOAD**, så det kræver en del **tilpasning**, hvis man vil anvende den **relationelle model**.
- **CASE-værktøjerne er dyre** at anskaffe

Overblik og sammenhænge for UML (Klaus' model)



Datamodeller (Y)

Bruges generelt	At modellere datastrukturer Få en forståelse af sammenhænge Velegnet til at modellere større databasebaserede edb-systemer		
	ER	UER	Klasse
Fulde navn	Entity relation	Uniform Entity Relation Uniform = de givne begrebsdannelser altid modelleres på én og kun én måde.	Class diagram
Bruges ved	Relationel db Hierarkisk db Netværks db	Samme som ER bare mere formalistisk	Objekt orienteret
Modeltype	Konceptuel ³	Konceptuel ³	Konceptuel (Analyse) Specifikation (Analyse/Design) Implementeringsklar (Design)
Anvendelse	<ul style="list-style-type: none"> • SimPLICITET • 	Velegnet til endelig systemspecifikation til implementering	Velegnet til komplekse systemer
Ulemper	<ul style="list-style-type: none"> • Egner sig ikke til store komplekse systemer 	<ul style="list-style-type: none"> • Ufleksibelt pga. den store grad af formalisme • Ukendt udenfor HHÅ 	<ul style="list-style-type: none"> • Kan være svært at overskue • Meget at huske på • Svær for nye at vurdere
Fordele	<ul style="list-style-type: none"> • SimPLICITET • Overskuelighed • Nem metode til at angribe • Velkendt • Standard 	<ul style="list-style-type: none"> • Prædikater: Sikrer mod misforståelser i sammenkædninger • Formalisme i højsæde • Nemt for andre at sætte sig ind i • Kan ret let overføres til programmeringskode • Forekomstgrafer • Medtage historik (temporale på en "god måde") 	<ul style="list-style-type: none"> • Objekt orienteret fordele <ul style="list-style-type: none"> ○ Indkapsling ○ Nedarvning (findes i ER) ○ Genbrug ○ Udvidelsesvenligt • Meget detaljeret • God ved komplekse systemer • "De nye programtyper" <ul style="list-style-type: none"> ○ CAD ○ CASE • Meget udbredt • Standard • Får meget med i diagrammet og kan bruges ret konkret i kodemæssig sammenhæng

Er modellens indhold mod klasse diagrammets indhold

Er-diagrammet	Klassediagrammet	Fordele
Entitet	Objekt	
Entitetstype	Klasse	
ISA	Nedarvning (Specialisering/generalisering)	Genbruger superklassen / entitetstypen i subclasserne / de underlæggende entitetstyper

ER:

En model der opfatter verdenen som bestående af basale objekter, kaldet entiteter og relationer mellem disse objekter.

- Entitetstype
 - Entitet er et objekt der eksisterer og som kan skelnes fra andre objekter, f.eks. en specifik person, virksomhed eller begivenhed
 - Beskrives via attributter

³ betyder at den viser begreber som kan henføres til de omgivelser, som modellen skal indeholde informationer om.

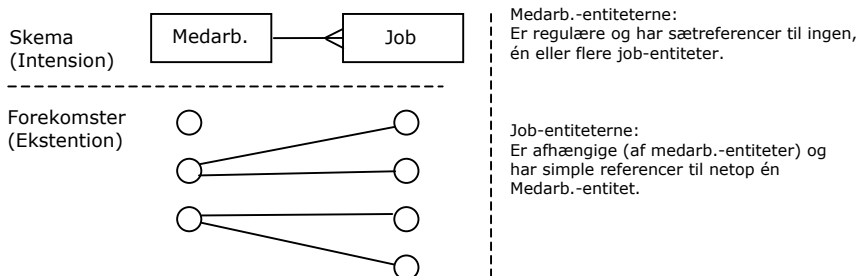
- For hver attribut er der et domæne, dvs. et område indenfor hvilket attributten må antage en værdi, f.eks. attributten postnr. må antage værdierne 1000-9999.
- Attributter kan karakteriseres via følgende attributtyper:
 - Simple, dvs. kan ikke opdeles
 - Composite, dvs. kan opdeles, men er ikke blevet det, f.eks. "navn" kunne have været opdelt i: Fornavn, Mellemnavn, Efternavn
 - Single valued, dvs. kan kun antage én værdi, f.eks. en persons postnr.
 - Multi valued, dvs. kan antage flere værdier, f.eks. en persons tlf.numre.
 - Derived, dvs. værdier, som man kan udlede noget af, f.eks. ud fra CPR.nr. kan der udledes fødselsdato og alder
- Relation
 - En relation er en association mellem flere entiteter
 - Et relationsæt er et sæt af relationer af samme type mellem 2 eller flere entitetsæt
- Begrænsninger
 - Kardinaliteter
 - En-til-en
 - En-til-mange
 - Mange-til-en
 - Mange-til-mange
 - Participating
 - Total relation: Hvis hver entitet i et entitetsæt er relateret til mindst en entitet i et andet entitetsæt. Eks. Et lån skal have en kunde...
 - Delvis (partiel): Hvis ikke alle er relaterede. Eks. En kunde kan have et lån
- Elementer i et ER diagram

Symbolerne	Repræsenterer...
Rektangler	Entitetsæt
Ellipser	Relationssæt
Linier	Relationen ml. relationssættet og entitetsættet
Dobbelte ellipser	Multi valued attributter
Stiplede ellipser	Afledte attributter
Dobbelte linier	Total relation
Dobbelte rektangler	Svagt entitetsæt (mangler primær nøgle)
Understregning i en ellipse	Nøgle

- Specialisering
 - Vises som en triangel, hvori der står ISA
 - Undergrupperne (subklasserne) nedarver fra overgruppen (superklassen)
- Generalisering
- Modsat af specialisering
- Reduktion af ER skema til tabeller
 - Hver entitetsæt og hver relationssæt får en tilsvarende tabel.
 - Et stærkt entitetsæt bliver til en tabel med de samme attributter
 - Composite attributter opdeles til flere attributter
 - Multi-valued bliver til en ny tabel
 - En svag entitetsæt bliver til en tabel hvori man inkluderer primærnøglen fra det stærke entitetsæt
 - En mange-til-mange relation repræsenteres med en ekstra tabel hvori de to tabellers primær nøgler er, samt attributter der relateres til relationen.
 - Ved mange-til-en relationer og en-til-mange relationer vælger man at droppe relationstabellen og i stedet indsætte en ekstra attribut i "mange"-tabellen, hvori primærnøglen fra "en"-siden er.
 - Ved en-til-en relationer har man frit valg til at vælge en af tabellerne som være "mange"-tabellen
 - Hvis man sætter dem sammen men der er tale om en delvis (partial) relation, så vil man kunne få null-værdier
 - Relationstabeller mellem en stærk og svag entitetsæt kan droppes
 - Specialisering, metode 1: Laves ved at oprette en tabel for høj-niveau tabellen og for hver lav-niveau tabel, og så have høj-niveau nøglen i hver lav-niveau tabel. Ulemper er at man skal ind i to tabeller for at hente information om en entitet.
 - Specialisering, metode 2: Laves ved at oprette hver lav-niveau tabel, og så lægge alle høj-niveau attributterne ned i lav-niveau tabellerne. Ulempen er muligheden for redundans, hvis man har en entitet der benytter både høj-niveau og flere lav-niveau tabeller. Eks. en ansat som også er kunde i virksomheden

- Aggregering: Lav en tabel som indeholder primær nøglen fra den aggregerede relation, primær nøglen fra den associerede entitetsklasse samt hvis relationen har nogle attributter, så også disse. En form for mange-til-mange tabel, bare med en nøgle for en anden tabel som kan være null.

UER:



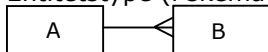
- **Entitet (i forekomstgrafen)**

En cirkel = node

- **Relationerne (i forekomstgrafen)**

Kanter

- **Entitetstype (i skema)**



Regulær = Kan eksistere uden B

Afhængig = B kræver A

A har sætreferencer til B

B simpel reference til A

- **Relation**

Disse kan også navngives

- **Entitetstypernes attributter**

Vises i kasser med en prik i toppen, el. som en streg ud fra entitetstypen

Følgende regler for forekomstgrafen:

- Refleksion: $a=a$
En node er relateret til sig selv
- Symmetri: $a=b \rightarrow b=a$
- Transitivitet: $a=b$ og $b=c \rightarrow a=c$

Faste sammenhængs-beskrivelser → Relaterede til mønstre:

Simpel sammenhæng A-<B
 Konjunktiv sammenhæng: A--C
 Disjunktiv sammenhæng: A>-B-<C
 Transitivt løb: A>-B>-C
 Cirkulær: A-<C-<B-<A

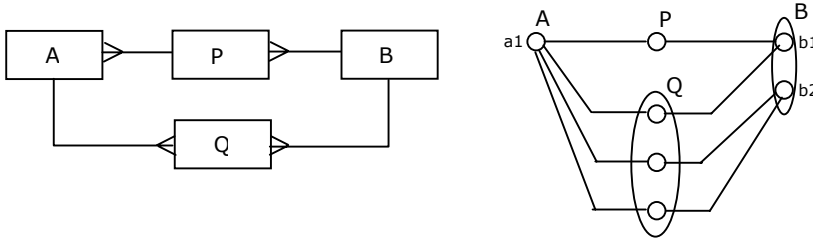
Lukket struktur

En række mange-mange rel. sat sammen i en lukket cirkel

Stiudtryk og prædikater:

- Stiudtryk
sekvens af variable svarende til entitetstyper, hvorimellem der er bestemte sammenhænge
- UER er ikke præcist nok når man f.eks. har flere matadorspil og en brik skal høre til et bestemt spil. Dette kan løses på 2 måder:
 - Prosa, dvs. skrive sig ud af det, F.eks. Brik skal være på felt, som tilhører et bræt, der er på et matadorspil, som brikken også tilhører
 - Prædikater, F.eks. EXIST //hmm...hvorfor EXIST?? – Hvis det havde været et referancekrav, så skulle det have været all – det mangler vi nok i vores opgave. EXIST er måske fordi, det kun er de brikker, der er i spil.... Hmmm... Nej jeg ved det ikke!!
Brik.Felt.Bræt.Matadorspil = Brik.Matadorspil

Stiudtrykstyper



Med A bundet til a1 løber stierne

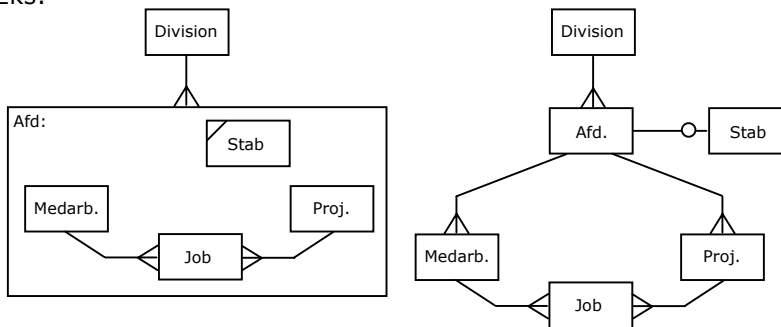
Simpel sti: A.P.B over: b1 (Udpeger kun en entitet, SQL: DISTINCT)
 Sætsti: A.Q.B over: b1, b2 (Udpeger flere forskellige entiteter, SQL: DISTINCT)
 Multisætsti: A.Q.Q.B over: b1, b2, b2 (Udpeger flere entiteter og også gentagelser)

Prædikater

- Prædikater = "Noget forud givet", dvs. en betingelse der altid skal være sand
- Handler om at følge stier
 - Man skal først se efter simple stier og dernæst sætstier
- Eks. Simple sti
 All Brik B1 as Brik and B2 as Brik
 B1.spiller = B2.spiller
 Imply
 Not(B1.Matadorspil = B2.Matadorspil)
 - Dvs. samme spiller har 2 brikker i ét bestemt matadorspil, altså en spiller kan have flere brikker, men det må ikke være i samme matadorspil.

Aggregering

- En bestemt måde at modellere på
 - Eks:



1. Referencekrav, Job:
 ALL Job (Job.Medarb.Afd=Job.Proj.Afd);

- Er en fordel ved modellering af større systemer, pga. forsimpningen.

Klassediagram:

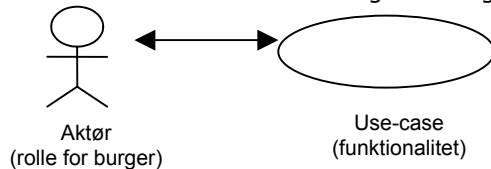
Se klassediagram som emne

Use-case og sekvensdiagram (Y)

Use-case

Use-case definitioner:

- Use Case [class]: *The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.* (UML definition)
- *A Use Case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.* (Alistair Cockburn, 1999)
- At lave en funktionel afgrænsning af systemet, som alle kan forstå (DC-def)



Formål med use-case (i kravspecifikation):

- At identificere og afgrænse funktionelle krav
- At specificere funktionelle krav på en overskuelig måde
- At organisere systemet set ud fra en brugers syn
- At beskrive, hvad man kan gøre med systemet
- Kravspecifikation er essentielt, da en ordentlig afgrænsning og forståelse af problemområdet, medfører en bedre måde at tackle systemet på.
 - At finde fejl på dette tidspunkt i processen er det optimale, da der så ikke er mange ændringer der skal foretages

Aktør:

- Aktøren har en anvenderrolle i forhold til det system der betragtes
- En aktør kan være en person eller et system
- En aktør navngives med rollenavnet (navneord) i ental og beskrives kort

Beskrivelse af use-case (Pre-/post conditioner)

- Use cases repræsenterer samlinger af scenarier med et fælles formål
- Definition (forklaring af funktionalitet)
- Pre-conditioner = Betingelser, der skal være opfyldt før use-case
- Post-conditioner = Effekter, efter use-case

Scenarier og use-cases:

- Et scenarie er et muligt gennemløb af en use-case, en brugssituation
- En use-case understøtter en arbejdsgang og kan dække over et eller flere konkrete situationer eller gennemløb
- En use-case har som hovedregel
 - **primær scenarie:** et "standardgennemløb" der resulterer i at aktøren får løst sin opgave (når sit mål)
 - **sekundære scenarier:** et eller flere alternative gennemløb der ofte håndterer fejlsituationer eller varianter af standardforløbet
 - **Fejl scenarier**

Udvidede use-cases (analyse)

- er mere detaljerede use-cases, idet de generelle use-cases fra kravspecifikationen opsplittes i mindre funktioner.
- Ret formalistisk
- Yderligere principper fra OO tilføjes i analysen:
 - **Nedarvning på use-case:** = Hele beskrivelse (def., pre/post, scenarier) + Tilføjer nye egenskaber
 - **Nedarvning på aktør:** = definition af rolle + relation til use-cases + udvide aktør definition
 - **Aggregering på use-case:**
 - **Include:** Use-case består af del use-cases
 - **Extend:** Use-case kan udvides med andre use-cases. Dvs. kan godt fungere uden de andre use-cases, men der kan tilføjes flere (i et evt. scenarie).

HUSK

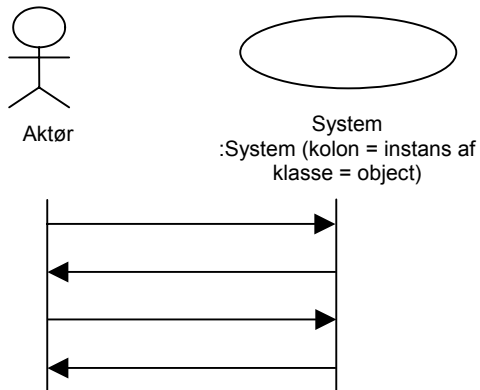
Sammenhæng mellem detaljerede use-cases og sekvensdiagrammer (både blackbox og design), da der skal være et sekvensdiagram for use-case scenarierne

Sekvensdiagram

- Give en forståelse af sammenhænge via meddelelserne sendes tidsafhængigt.
- Et essentielt link mellem use case stier og koden

Blackbox sekvensdiagram (analyse)

- Viser interaktion mellem aktør og system (for et scenarie), samt i hvilken rækkefølge denne foregår.
- Pilene mellem aktør og system viser input/output til/fra systemet i form af data og funktionskald.
- Sammenhæng med:
 - use-case og scenarier og selvfølgelig sekvensdiagrammet i design.



Design sekvensdiagram

- Hører til gruppen af **interaktionsdiagrammer**
- Her er **tidsperspektivet** altså vigtigt, dvs. rækkefølge af interaktionen mellem aktør og klasser og de anvendte metoder.
- Da klassediagrammet i design ofte bliver komplekst mht. til metoder og hvilke instantierede klasser (objekter), der kan kalde metoderne, så er sekvensdiagrammer gode til at **give overblik** over de anvendte metoder ved en given interaktion med systemet.
 - Dette vil også lette selve programmeringen, da sekvensen og dataanvendelsen således allerede fremgår.
- **Spezialklasserne** fra klassediagrammet indgår ligeledes, og disse er:
 - **GUI** (grafisk eller ej)
 - Al interaktion fra aktør til system foregår via GUI-klassen
 - Anvendes fordi brugerne af systemet er forskellige og derfor skal have adgang til forskellige interfaces afhængig af tilgængelige funktioner.
 - **Controller**
 - GUI-klassen sender og modtager meddelelser til controller-klassen, dvs. denne uddelegerer de opgaver som GUI kalder.
 - Er et designmønster tilhørende GRASP
 - Evt. kan man nøjes med kun en controller-klasse, men ofte er der flere (use-case controllers), fordi brugerne kun skal have adgang til relevante funktioner (bl.a. af sikkerhedsmæssige årsager).
 - **Container (hjælpe-klasse)**
 - En spand med objekter
 - Bruges når man skal **finde** et bestemt objekt eller flere, f.eks. hvis man har et navn og ønsker alle person objekter med det navn
- **Metoderne mv.**
 - En metode returnerer en "værdi" (**returværdi**), der kan være en alm. datatype (tekst, tal, dato) eller en komplekstype, f.eks. et objekt eller en vektor (flere objekter?)
 - En metode kan være iterativ, dvs. udføres flere gange, hvilket markeres med stjerne (*). Dette har vi glemt i vores diagram.
- Kan også overføres til en **ER-udgave**
 - Sammenhæng ml. gui, controller og datamodel (databasen)
 - Sender i stedet SQL-sætninger

Temporale modeller (Y)

- Hvad er temporalitet - forskellige **systemers håndtering af tid**:
 - **OLTP** (OnLine Transactional Processing)
 - Alm. programmer (bl.a. relationel model, OODB, alm. UER)
 - Modellerer som oftest virkeligheden som den ser ud **nu** og her.
 - **OLAP** (OnLine Analytical Processing)
 - Data Warehouse (stjernemodellen)
 - Formålet: Registrere + analysere **historiske data**
 - **Temporale datamodel**
 - vise virkeligheden set **over tiden** (dvs. både nutid og fortid – sågar fremtiden kan "planlægges").
 - Ofte anvendes den bitemporale konceptuelle datamodel (eng.: Bitemporal Conceptual Data Model = **BCDM**), der minder kraftigt om den **relationelle model**.
 - **BCDM** er en simpel model, der ikke fokuserer på præsentation, lagring eller evaluering af forespørgsler. Fokus lægges på høj performance.
- Tid i databaser - To **tidsbegreber** (primært møntet på BCDM):
 - **Valid Time (VT)**
 - Viser en tidsangivelse for hvornår data er/var gældende (relevante) i databasen
 - Fx gælder en recept kun i en begrænset tidsperiode men kan godt udstedes på et tidligere tidspunkt end gyldighedsdatoen.
 - En speciel attribut i en relation (tabel), der kan indeholde flere tidsmængder (multimængde attribut). VT{BeginT;EndT}
 - Mest interessant
 - **Transaction Time (TT)**
 - Hvornår data eksisterede i DB
 - Data er både fysisk og logisk eksisterende, når de er indenfor TT. Logisk når data er i den konceptuelle DB (= man kan se data i en tabel på tidspunkt t), fysisk når de ligger på disken.
 - Starttid = Oprettelse af tuple
 - Sluttid = Når den logisk slettes, men fysisk skal blive liggende
 - **Bi-temporal relation**: Hver tuple gemmer både VT og TT.
 - Placerer tidsaspekt i et VT og et TT felt, da disse er multi-valued attributter (kan nøjes med VT og kan også opdele det i en starttidsattribut og sluttidsattribut)
 - From: 1/1/2003 9:00 + To: 1/4/2003 16:00
 - Eller VT: {1/1/2003-9:00; 1/4/2003-16:00}
- **Fordele**
 - Mulighed for at gemme historiske data + referencer, og skrue tiden tilbage for at se data på et bestemt tidspunkt (snapshot).
 - Mulighed for at planlægge ud i fremtiden, f.eks. at patienten kun skal have et medikament til januar 2004.
 - Lave forespørgsler på tværs af tid (TQL, TSQL2) = vigtig fordel!
- **Ulemper / Problematikker**
 - DB'en fylder oftest meget mere end i den relationelle model
 - → Forespørgsler tager længere tid
 - Man får problemer, hvis der ikke er tidsmæssig konsistens (fx huller i tiden hvor man ikke kan se reference for en entitet)
 - UER: tiden i mange-relationen skal være indeholdt i en-relationen (evt. på flere entiteter)
 - UER: hvis reference mellem entiteter skifter, så kan det give problemer med tidshuller – Løsning er at gøre referencen temporal
 - Da det er noget dyrere at udvikle en temporal model, så skal klienten virkelig have brug for det før end at det økonomisk kan betale sig.
- Et tekstuel **eksempel**
 - Fremtidige / historiske data
 - transaktioner på en bankkonto
 - historik på ansattes tilhørsforhold til afdelinger
 - Fremtidige data
 - d. 1/10 besluttes det, at Hansen skal overføres pr. 1/11 til en ny afdeling (Økonomiafdeling)

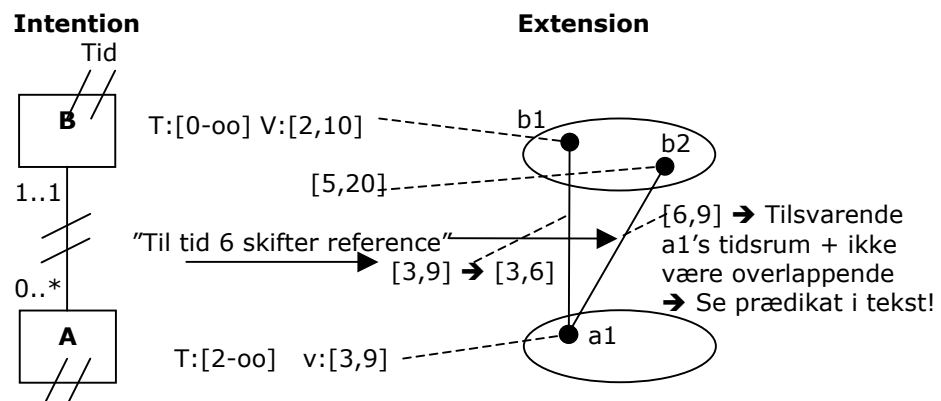
- Kan derfor gemme to oplysninger om personen nemlig hr. Hansen før d. 1/11 og fra den 1/11, og i afdelingsoplysningerne skal vi ligeså have oplysninger fra før 1/11 og en tuple i øko. afdelingen som er gældende fra d. 1/11
- Ellers må vi nøjes med at foretage ændringen til den pågældende tid

- **Temporale forespørgselsprog**

- **TQL** (Temporal Query Languages)
 - Udviklet for at simplificere modelleringen af tid og tids relaterede forespørgsler.
 - Har mange muligheder i stil med Relational Algebra: intersect, union, selection, projektion, join
 - Muligheder
 - **SNAPSHOT**
Giver mulighed for at se data på et helt bestemt tidspunkt (= bestemt VT). Man stiller således tiden tilbage, og kan altså se data, som de så ud på dette tidspunkt.
- **TSQL2** (Temporal Structured Query Language) – bygger på **BCDM**
 - Ikke med i standard SQL92, men som en tilføjelse
 - Direkte temporalt modul i nyeste SQL-standard SQL:1999 (SQL3?)
 - Som hovedregel bruges det kun til forespørgsler over tid, men ønsker man en alm. forespørgsel anvendes SNAPSHOT.
 - Datatyper:
 - Fra SQL-92: DATE, TIME, TIME-STAMP og INTERVAL
 - Nyt i TSQL2: PERIOD = faste tidsintervaller, f.eks. år, dage og minut

Håndtering af temporalitet i forskellige datamodeller:

- **UER** (ER indeholder som standard ikke temporalitet)
 - Markeret ved to parallelle streger
 - Entitetstyper og referencer kan være temporale



- **OO Klassediagram (UML)**

- Ingen speciel markering, men evt. kan man anvende notationen fra UER.
- Til at løse problemet med at gøre datamodellen temporal, så har OO-guruen Martin Fowler samlet en række **designmønstre** for temporale elementer i OO-sammenhæng
 - **Effectivity**: Actual time (valid time) for et object – typisk ved associeringer
 - **Snapshot**: Et view på objektet på et bestemt tidspunkt
 - **Temporal Object**: Et objekt, der ændrer sig over tid (fuld temporalt)
 - **Temporal Property**: En attribut, der ændrer sig over tid
- **Til vores opg.:**
 - temporale klasser/objekter
 - Fly: temporal property
 - Bestiller: snapshot eller temporal objekt
 - temporale associationer (Flyveplan ----- Fly)
 - Effectivity → ny klasse (fly-anvendelse) på associering til det temporale

Objekt orienterets fordele/ulemper (Y)

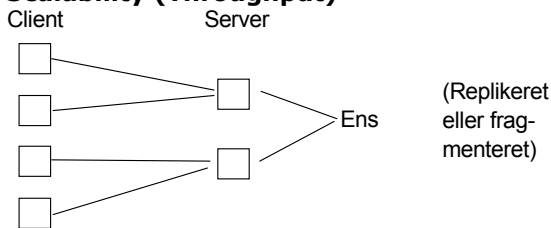
Husk at en klasse svarer til en type. Eks. vil noget så simpelt som int, svare til en klasse kaldet int med en attribut og en række metoder. Dvs. at man i OO verden kan definere alle de typer kan har lyst til.

- **Kendetegene** ved objekt orienteret
 - **Klasser (Classes)**
 - Er en statisk skabelon der indeholder den definerede struktur (attributter) og adfærd (metoder) for en entitet i applikationsdomænet.
 - Når en klasse instantieres bliver et objekt født som et billede af klassen.
 - Er således en beskrivelse af en mængde af objekter med de samme egenskaber.
 - **Komplekse typer (Complex types)**
 - Frem for simple typer som boolean eller string, kan man lave typer som en ordre og en kunde.
 - **Indkapsling** (Encapsulation / information hiding)
 - At objektet stiller et offentligt interface til rådighed som giver adgang til objektets attributter og metoder, men som til gengæld skjuler alt andet derunder kodningen.
 - Metodens kode kan ændres uden at det offentlige interface ændres. Dette medfører at andre programmer ikke skal ændres hvis objektet ændres.
 - Utsigtede sideeffekter kan undgås
 - **Nedarvning** (Inheritance)
 - At en klasse (subklasse) arver dets overordnede classes (superklasse) attributter og metoder
 - Message-passing
 - At objekter samarbejder ved at sende beskeder til hinanden.
 - **Polymorfisme** (Polymorphism)
 - At flere objekter af forskellige (sub)klasser kan have metoder med samme navn og betydning men hvad den enkelte metode gør kodemæssigt kan være forskelligt klasserne imellem.
 - Ek.s kan det udnyttes i en senere forespørgsel, hvor man er ligeglad med hvilken subklasse det er, da vi jo bare kalder den samme metode (BeregnRente eks,)
- **God ved**
 - God ved høj systemkompleksitet, men som ikke består af adm. systemers db. Eks. CAD systemer.
 - **Udvidelsesvenligt**
 - Skal ikke ændre i hele systemet, kun de dele der skal udvides.
 - Indkapsling, nedarvning og polimorfisme hjælper på dette
 - **Genbrug**
 - Af klasserne
- **Dårligt**
 - hvis man har store datamængder i en container og skal spørge alle om de fx hedder 'jensen'
 - Simple systemer er det nok ik så relevant ved
 - **Ik så nemt at gå til**
- **Eksempler**
 - Y2K
 - Ved relationel model, ændrer man i db'en og skal dernæst ændre i alle progs som benytter db'en
 - Ved objekt orienteret ændrer man i klassens attributter og måske metoder, men udadtil er der ingen ændringer i forhold til andre programmer

Datadistribution (Y)

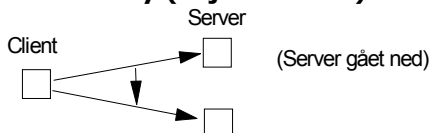
- **Årsag** til at man skal tænke på datadistribution er

- **Scalability (Throughput)**



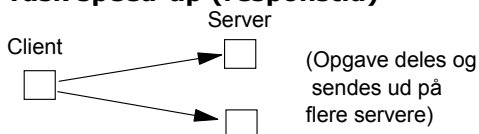
- Udvidelsesvenligt system
 - Flere servere, helst replikering

- **Availability (Fejltolerance)**



- At man sikrer adgang til data
 - Replikering

- **Task speed-up (responstid)**



- Opgaver uddeles ml. flere servere for at blive udført hurtigere
 - Replikering

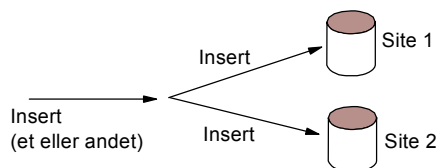
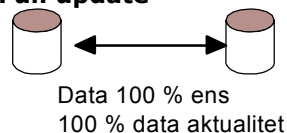
- Lige et mellem**eksempel**

- Flyselskab har en afdeling i Tyskland og DK.
 - Ønske om at forbedre datadistribution:
 - Scalability
 - Flere kunder kan komme ind på hjemmesiderne
 - Availability
 - Går en ned, ryger hele systemet ikke, "kun" et land. Dog vil ssh. for 1 går ned være større
 - Minimerer også netværksforbrug
 - Kræver parallellisme
 - Fragmentering
 - Fordel at lave vertikal fragmentering, ml. dk og d kunder.
 - Replikering
 - 100 % dataaktualitet ikke problem med opdateringer

- **Opnås** gennem **parrallellisme**

- Samtidig arbejde på data
 - Opnås gennem
 - **Replikering**
 - + Tilgængelighed
 - + hvis kun læses, kan ting ud føres parrallelt

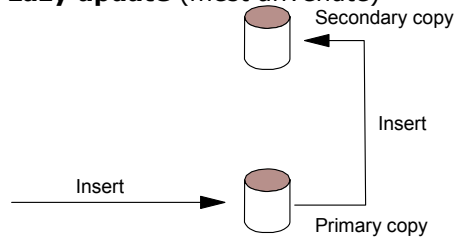
- **Full update**



Man må ikke læse den ene, mens site 2 opdateres
Man kan bruge lock ligesom ved transaktioner

- Transaction_begin
 - Insert på site 1
 - Insert på site 2
- Commit

- **Lazy update** (mest anvendte)



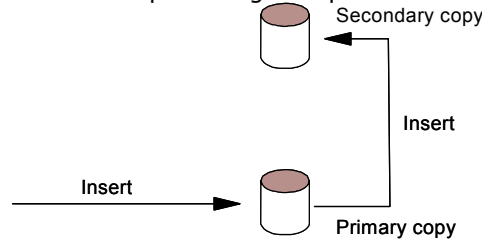
Næsten 100 % dataaktualitet

Primary copy
Read / write
Secondary copies
Read

Konflikter i updates, hvis een inserter på secondary copy samtidig med, at der insertes på primary copy

- **Snapshot**

Periodevis opdatering af replikerede data



(Udføres kun een gang i døgnet, eller en gang om ugen ((Passende interval)))

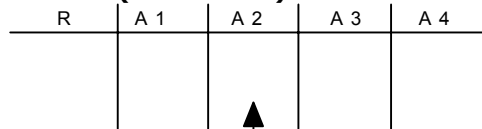
Ikke så

vigtigt om det slår igennem med det samme (eks. banker får først en adresse ændring fra personregistret efter et par dage)

Fordel ved snapshot er, at det kan opfylde ønske om at skabe sikkerhed omkring primary copy, så der kun gives adgang til secondary copy.

- **Fragmentering**

- **Vertikal (Attributter)**

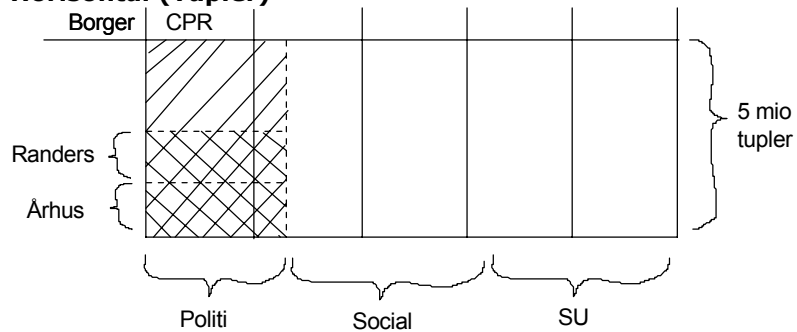


Til brug for join

F1: $\Pi_{A1, A2}(R)$
F2: $\Pi_{A3, A4, A2}(R)$

Komplethed: Alle attributter skal deltage i et fragment
Rekonstruere: F1 |X| F2

- **Horisontal (Tupler)**



F1: $\sigma_{\text{land}=D}(\text{Kunde})$

F2: $\sigma_{\text{land=DK}}$ (Kunde)

Komplethed: Alle tupler skal tilhøre et fragment

Rekonstruktion: F1 U F2 (Foreningsmængde)

- **Vores opgave:**

- Scalability – vigtigt, da vi forventer mange brugere af systemet
 - Medfører **replikering** af DB
 - I starten full-update, evt. senere snapshot update (periodevis)
 - Brugere skal arbejde på sekundær copy, bl.a. af sikkerhedsårsager
- Availability – meget vigtigt at servere ikke går ned, dvs. replikering sikrer, at vi kan fortsætte på en anden server (med samme DB), såfremt en server crasher.
- Task-speed up – har vi ikke nævnt!
- Evt. også **fragmentering**, da systemet er internationalt
 - F.eks. Horizontalt for bestiller, adm. personale og fly.
 - Fordel: **minimere forbrug af netværksressourcer**
- Ovenstående er de 4 (3) årsager til (fordele ved) parallellisme, som vi søger at opnå med vores system.

Teknisk arkitektur (Y)

- Design

- Centraliserede systemer

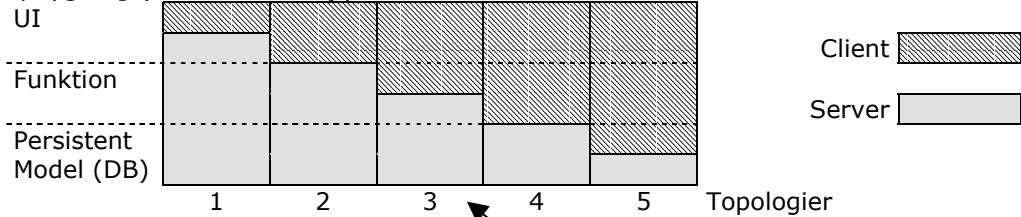
- Single user PC (uden kommunikation) → Disk. Denne type forsvinder nu.
 - Mainframes (eksisterer stadigvæk, men hedder nærmere en high performance server)
 - Transaktionsorienteret (typisk flere mio. i sekundet)

- Distribuerede systemer

- Client/server (traditional)
 - Webbaseret (også client/server)
 - Peer-to-peer

- Client/server modeller

- Opbygning (Gartner Group)



1: Gamle mainframes (centraliseret!)

1: Smpel web

Client = Browser (HTML-kode)
Server = HTML + Funk. + Model

3+4: De fleste CS-systemer

Nogle webbaserede

Client = Browser + scripting + java applet + .net

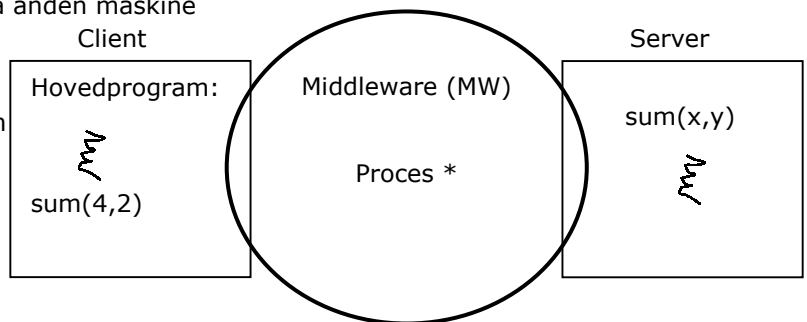
Her mener vi, at vi er i opgaven

- Protokol

- = "Hvordan skal enhederne opføre sig overfor hinanden"
 - Client (aktiv)
 - Server (passiv)
 - Meget simpel protokol
 - Det går aldrig galt
 - Mest fremherskende (altså traditionel client/server)

- Remote procedure call (i C/S via MW)

- Fange procedurekald
- Finde procedurekald på anden maskine
- Sende parametre
- Udfører sum
- Returnere resultat
- Aktivere hovedprogram

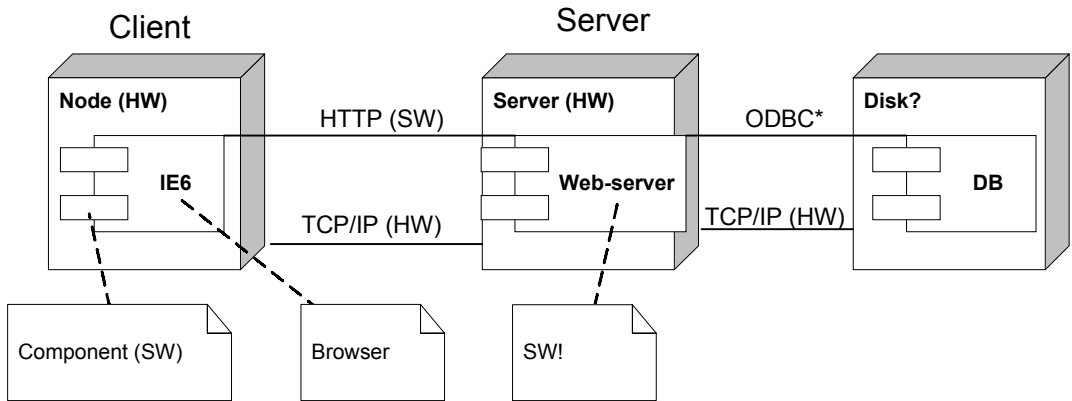


- Middleware (MW)

- Man kan implementere mange ting "her" (dvs. i MW-cirklen), f.eks. sikkerhed
 - Aner ikke om det er lokal eller global procedure
 - Allokere efter hvad der er optimalt (til enten server eller client).
 - Typer af MW:
 - CORBA** standard: **Forskellige sprog kan kommunikere**, ex. Hovedprogram i delphi og procedure i C++
 - RMI** (Remote Method Invocation) - java
 - DCOM** (Microsoft)
 - Internet** er MW (både på client og server)
 - Nameserver (DNS)
 - webserver
 - proxyserver (sikkerhed)

- **Webbaseret system**

- Deployment diagram (UML) (også vores...!)



-

3-tier model:

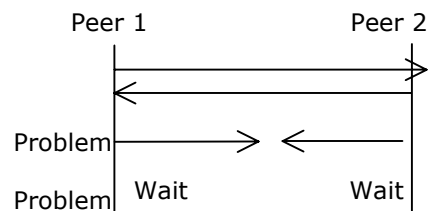
UI	Client
Funktion.	Appl. server
DB (model)	Server

- Dette web-baserede system er en kombination af topologi 3) + 4).
- **Web-teknologi:**

Browser		Server		DB
Nutid:				
- Fortolke HTML (statisk)		MS-webserver Apache server		- Relationelle - OO
Funktioner: - Scripting - Java applets - Java objekter - .net	HTML ←	Funktioner: - Scripting * Objekter: - JSP ** - .net	ODBC JDBC *** ADO (MS) ↔	- Objekt/ relationelle
Fremtid:				
Erstatter HTML: - Stylesheets (CSS) - XML - Java - .net	XML ←	- JSP ** - .net	XML ↔	XML

- **Peer-to-peer**

- Alle har same rolle
 - kan starte kommunikation
 - kan stoppe kommunikation
 - Protokol: TCP/IP løser konflikter
 - f.eks. Kazaa


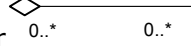


- **Vores opgave:**

- Webbaseret client/server
- Topologi 3, dvs. lidt funktionalitet på klient (scripts)
- ASP eller PHP + HTML.
- http-protokol (SW) + TCP/IP (HW)
- Dataadgang via ODBC.
- Evt. XML i fremtiden.

Klassediagram & (Y)

Klassediagram (kernen i UML)

- Formål: Modellerer objekter (hvor man i ER modellerer entiteter) mht. klasser (typer), attributter, metoder og associationer mellem klasser.
- **Analyse** diagram
 - **Struktur** er vigtigt → Fokus på: analyseklasser, objekter og associeringer
 - **Associeringer**
 - Alm. (ligesom i ER), 1-1, 1-M og M-M
 - Aggregering
 - Stærk (composite) = Består af  1 0..*
 - Svag = har  0..* 0..*
 - Specialisering/generalisering
 - Superklasse (øverst): Er den abstract (= har ingen objekter)?
 - Subklasse (nederst): Nedarver attributter og metoder fra superklasse
 - disjoint (kan kun være en) vs. overlapping (kan være flere subklasser) --- complete (superklassen er abstrakt) /incomplete (man kan nøjes med at være superklasse)

Design diagram

- **Detaljer** er vigtigt → Fokus på: metoder, attributter og specialklasser (UI, controller, container mv.)

Metoder:

- Alle skal med. brug evt. sekvensdiagrammet til at sikre dette.

Attributter

- Alle skal med.
- Visibility (hvem kan se attributter – indkapsling...):
 - Indefra, udefra, nedarvning

Navigation

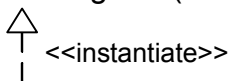
- X kan se (kender til eksistensen af) Y, men ikke modsat!



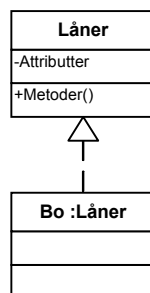
Specialklasserne:

- **GUI / UI / I** (grafisk eller ej)
 - Anvendes fordi brugerne af systemet er forskellige og derfor skal have adgang til forskellige interfaces afhængig af tilgængelige funktioner.
- **Controller**
 - GUI-klassen sender og modtager meddelelser til controller-klassen, dvs. denne uddelegerer de opgaver som GUI kalder.
 - Er et designmønster tilhørende GRASP
 - Evt. kan man nøjes med kun en controller-klasse, men ofte er der flere (use-case controllers), fordi brugerne kun skal have adgang til relevante funktioner (bl.a. af sikkerhedsmæssige årsager).
 - I controlleren er desuden alle metoder fra de klasser denne er associeret med.
- **Container (hjælpe-klasse)**
 - En spand med objekter
 - Bruges når man skal **finde** et bestemt objekt eller flere, f.eks. hvis man har et navn og ønsker alle person objekter med det navn

Klassediagram (intension)



Objektdiagram (extension)



Objektdiagrammet ligner meget klassediagrammet.

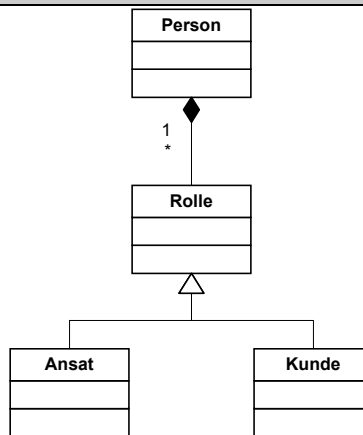
Mønstre (Y)

- **Definition:** Et mønster er en generel løsning til et generelt defineret problem.
- **Fordel:** Letter modelleringen...(givetvis også implementeringen)
- **Standardmønstre (analyse)**
 - Disse kommer fra uddraget af den danske UML bog [OOA&D]. Jeg har valgt at inddele dem i to grupper, selvom kun den sidste gruppe er navngivet i bogen.
 - **"Strukturelle mønstre"** – bruges til at beskrive forhold mellem objekter
 - Rollemønstret (vigtigt)
 - Relateringsmønstret
 - Hierarkimønstret (vigtigt)
 - Genstand-Beskrivelsesmønstret (vigtigt)
 - **Basale adfærdsmønstre** – bruges ofte i forbindelse med tilstandsdiagram (= mønstre med tilstandsskift)
 - Trinvis-relation mønstret
 - Trinvis-rolle mønstret (vigtigt)
 - Samling mønstret (vigtigt)

Rolle-mønstret (strukturelt)

Udbredt mønster, hvis f.eks. personer skal have flere roller i systemet.

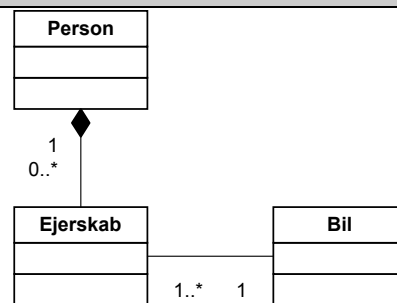
Fordelen er her, at alle de generelle (fælles) personoplysninger for ansat og kunde, f.eks. navn, adresse og telefon, kan rolleklassen fremskaffe fra attributterne i personklassen.



Relateringsmønstret (strukturelt)

Bruges når en associering mellem to objekter skal have nogen egenskaber. Her er det iflg. litteraturen bedre at bruge dette mønster end en associeringsklasse. Der er således tale om en "mange til mange" relation, hvor den ene del af mønstret er en aggregering.

Her kan en person eje 0 eller flere biler, men en bil kan gennem sin levetid have haft flere ejere. (Minder om et temporalt mønster, Effectivity).

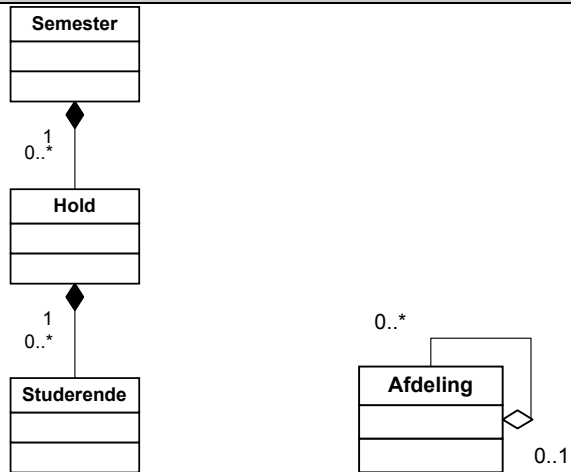


Hierarki -mønstret (strukturelt)

Meget anvendt mønster, idet en struktur således kan lagdes. Der findes en del undermønstre, da der er mange typer hierarkier i den virkelige verden.

Til venstre: **standardeksemplet** fra bogen:

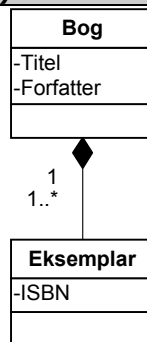
Til højre: Klaus' **Afdelingshierarki**, der er karakteriseret ved ens klasser i form af underafdelinger. Læg mærke til, at det er en svag aggregering!



Genstand-Beskrivelse -mønstret (strukturelt)

Simpelt mønster, der kan anvendes i systemer, der administrerer forskellige typer af beskrivelser med tilhørende eksemplarer (genstande), f.eks. kontrakter, bøger osv. Eneste krav er, at eksemplaret skal have en unik attribut som identifikation (ikke nødvendigvis en primær nøgle), f.eks. ISBN.

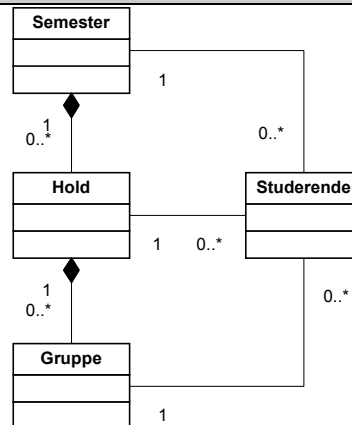
Her er eksemplet fra bibliotekscasen: Bog er her = beskrivelse, og eksemplar = genstand.



Trinvis-relation -mønstret (adfærd)

Bruges når objekter i problemområdet trinvist/sekventielt relateres til et hierarkisk mønster.

I følgende eksempel er der tale om oprettelse af en studerende, hvor denne først relateres til semester (øverste klasse i hierarkiet), dernæst hold og endelig læsegruppe. Således associeres Studenten med alle klasserne i hierarkiet – bare i en bestemt rækkefølge (sekvens).

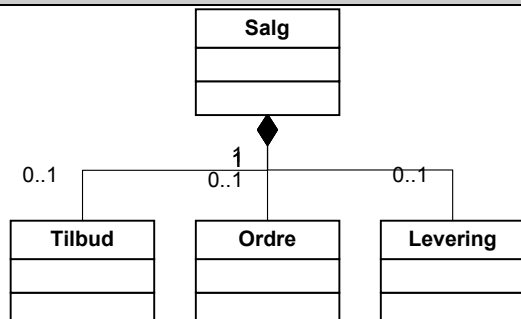


Trinvis-rolle -mønstret (adfærd)

Dette mønster minder en del om trinvis-relation og meget om den simple variant af rollemønstret.

Forskellen er blot, at rollen skifter sekventielt, dvs. når en rolle (tilstand om man vil) ændres, så har man ikke længere den tidligere rolle (og kan heller ikke komme tilbage dertil pga. sekvensen). Desuden kan man kun have en enkelt rolle af gangen.

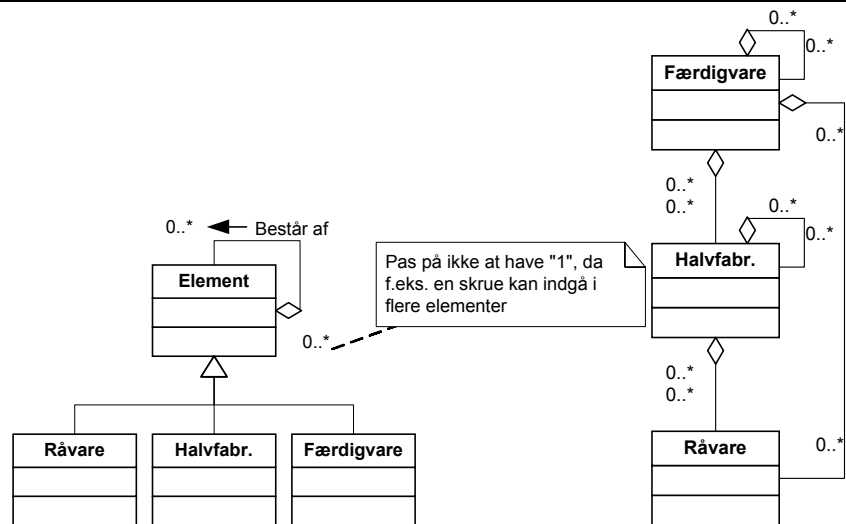
Et glimrende eksempel herpå er et salg, der i dette eksempel gennemløber 3 faser, der alle er en del af salget.



Samling -mønstret (adfærd)

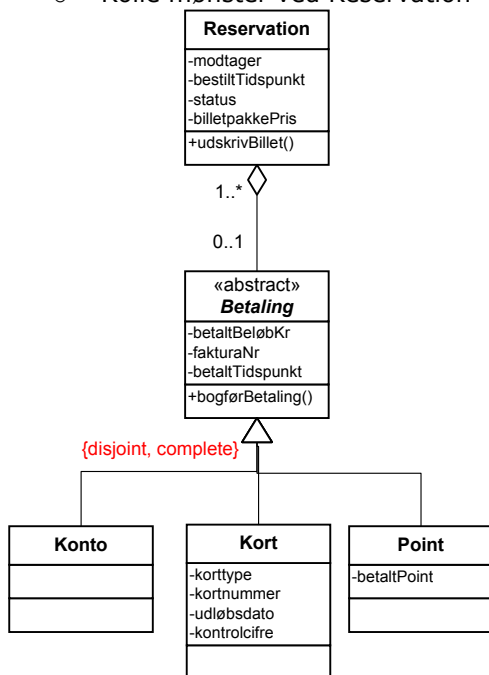
Anvendes typisk, hvis nogle dele består af andre dele (samling). Dvs. her er det typiske eksempel forholdet mellem **råvare-halvfabrikta-færdigvare** fra produktionsteorien.

Klaus viste to eksempler:
 TV: Blanding af specialisering og aggregering
 TH: Ren aggregering (svag), men mellem alle klasser. Her kan færdigvaren altså bestå af en blanding af andre færdigvarer, halvfabrikta og/eller råvarer.



Mønstre i vores opgave:

- Det vi huskede:
 - Rollemønstret ved interessant: Bestiller, passager og ansat
- Det vi glemte
 - Rolle mønster ved Reservation – betaling. Sådan burde det se ud...



- Desuden er der et Genstand-beskrivelsesmønster mellem Fly og flytype. Sådan burde det se ud...

